

Notes on Modern C++

(STL, Design Pattern, Concurrent, Template Programming)

Gyubeom Edward Im*

February 22, 2025

Contents

1	Introduction	6
1.1	What is new in modern c++	6
1.2	Table summary	7
2	Intermediate	16
2.1	Temporary	16
2.1.1	return-by-value vs return-by-reference	16
2.1.2	temporary and casting	17
2.2	Conversion	17
2.2.1	Conversion constructor, conversion operator	18
2.2.2	explicit constructor	19
2.2.3	explicit conversion operator	20
2.2.4	explicit(bool), c++20	21
2.2.5	Conversion example: nullptr	22
2.2.6	Conversion example: return type resolver	22
2.2.7	Lambda expression and conversion	22
2.3	Constructor	23
2.3.1	Base from member idioms	24
2.3.2	Constructor and virtual function	25
2.4	This call	25
2.5	Member function pointer	27
2.5.1	<code>std::invoke</code> , <code>std::mem_fn</code>	28
2.6	Member data pointer	28
2.7	Implement custom max	29
2.7.1	Add compare operator	31
2.7.2	c++20 ranges algorithm	32
2.8	Size of member function pointer	32
2.9	new, delete and placement new	34
2.9.1	Using placement new	35
2.9.2	vector and placement new	35
2.10	Trivial constructor	36
2.10.1	Trivial default constructor	36
2.10.2	Trivial copy constructor	36
2.11	Type deduction	37
2.11.1	Auto type deduction	38
2.11.2	Array Name	39
2.12	Rvalue & forwarding & reference	40
2.12.1	Lvalue vs Rvalue	40
2.12.2	Reference & Overloading	41
2.12.3	Reference collapsing	42
2.12.4	Forwarding reference	43
2.13	Move semantics	44

*blog: alida.tistory.com, email: criterion.im@gmail.com

2.13.1	Move constructor	44
2.13.2	std::move	45
2.13.3	Move and noexcept	45
2.13.4	Default move constructor	47
2.13.5	Rule of 3/5/0	47
2.14	Perfect forwarding	48
2.14.1	Using forwarding reference	49
2.14.2	Variadic parameter template	49
2.14.3	Member function pointer for perfect forwarding	50
2.14.4	Function object for perfect forwarding	51
2.14.5	Chronometry implementation	51
2.14.6	Notice for perfect forwarding	52
2.14.7	Perfect forwarding in STL	52
2.15	Callable object	54
2.15.1	Function object	54
2.15.2	Function object = function with state	55
2.15.3	closure & function object	56
2.15.4	inline & function object	56
2.15.5	STL & function object	57
3	STL programming	59
3.1	STL structure	59
3.1.1	Generic algorithm - find	59
3.1.2	Make iterator	61
3.1.3	Generic Algorithm - list	62
3.1.4	Container, Iterator, Algorithm (CIA)	64
3.1.5	Member type	65
3.2	Iterator	66
3.2.1	Iterator concept	66
3.2.2	Container iterator	67
3.2.3	Iterator invalidation	68
3.2.4	std::ranges::begin	69
3.2.5	Iterator value type	70
3.2.6	Iterator category	71
3.2.7	Iterator operation	73
3.2.8	<code>++</code> vs <code>next</code>	74
3.2.9	<code>std::copy</code> vs <code>std::ranges::copy</code>	75
3.2.10	Reverse iterator	76
3.2.11	Insert iterator	77
3.2.12	<code>std::counted_iterator</code> & <code>std::default_sentinel</code>	80
3.2.13	Iterator and sentinel	82
3.2.14	Ostream iterator	83
3.2.15	Ostreambuf iterator	83
3.2.16	Istream iterator	84
3.3	Algorithm	85
3.3.1	Erase-remove idioms	85
3.3.2	Algorithm vs member function	86
3.3.3	<code>std::erase</code> , <code>std::erase_if</code>	87
3.3.4	Algorithm with function as parameter	88
3.3.5	Predicate	88
3.3.6	Algorithm copy version	89
3.3.7	Projection	90
3.3.8	Constrained algorithm function object	91
3.4	Container	93
3.4.1	Container basic	93
3.4.2	Allocator	94
3.4.3	Sequence container	96
3.4.4	<code>std::vector</code>	96
3.4.5	Capacity	98
3.4.6	Container & user defined type	100
3.4.7	Emplace	101

3.4.8	<code>std::array</code>	104
3.4.9	<code>std::move (std::array)</code>	105
3.4.10	<code>stack, queue</code>	106
3.4.11	<code>priority_queue</code>	107
3.4.12	underlying container	109
3.4.13	<code>std::set</code>	110
3.4.14	find in <code>std::set</code>	111
3.4.15	<code>std::set</code> with user defined type	112
3.4.16	<code>std::less</code> implementation	114
3.4.17	<code>is_transparent</code>	115
3.4.18	<code>std::map</code>	117
3.5	Ranges	119
3.5.1	Range library	119
3.5.2	using views	120
3.6	Utility	120
3.6.1	<code>std::bind</code>	120
3.6.2	<code>std::function</code>	124
3.6.3	<code>std::reference_wrapper</code>	125
3.6.4	Smart pointer	128
3.6.5	<code>weak_ptr</code>	133
3.6.6	<code>unique_ptr</code>	135
3.6.7	<code>chrono (ratio, duration)</code>	137
4	Design Pattern	140
4.1	Separation of commonality and variability	140
4.1.1	Template method pattern	140
4.1.2	Strategy pattern	142
4.1.3	Policy base design	144
4.2	Recursive pattern	147
4.2.1	Composite pattern	147
4.2.2	Decorator	152
4.3	Principle of indirect layer	157
4.3.1	Adapter	157
4.3.2	Proxy	161
4.3.3	Facade	162
4.3.4	Bridge	164
4.4	Notification, enumeration, visitation	167
4.4.1	Observer	167
4.4.2	Iterator	169
4.4.3	Visitor	171
4.5	Creating and sharing objects	175
4.5.1	Singleton	175
4.5.2	Factory	178
4.5.3	Prototype	181
4.5.4	Abstract factory	183
4.5.5	Flyweight	184
4.5.6	Builder	187
4.6	MISC	189
4.6.1	Memento	189
4.6.2	State pattern	190
5	Concurrent programming	193
5.1	Warming up	193
5.1.1	Concurrent intro	193
5.1.2	<code>std::this_thread</code> namespace	193
5.1.3	Chrono	194
5.2	thread basic	195
5.2.1	<code>std::thread</code>	195
5.2.2	<code>std::ref, std::reference_wrapper</code>	198
5.2.3	<code>std::promise</code>	199
5.2.4	<code>std::future</code>	201

5.2.5	<code>std::packaged_task</code>	202
5.2.6	<code>std::async</code>	203
5.2.7	<code>std::jthread</code>	205
5.3	Synchronization	206
5.3.1	thread synchronization	206
5.3.2	<code>std::mutex</code> , <code>std::timed_mutex</code>	207
5.3.3	<code>std::shared_mutex</code>	210
5.3.4	<code>std::lock_guard</code>	210
5.3.5	<code>std::unique_lock</code>	211
5.3.6	<code>std::lock</code> , <code>std::scoped_lock</code>	213
5.3.7	<code>std::shared_lock</code>	214
5.3.8	<code>std::condition_variable</code>	215
5.3.9	<code>std::semaphore</code>	218
5.3.10	<code>std::latch</code> , <code>std::barrier</code>	219
5.3.11	<code>thread_local</code>	221
5.3.12	<code>std::call_once</code> , <code>std::once_flag</code>	222
5.4	atomic	224
5.4.1	<code>std::atomic</code>	224
5.4.2	reordering	227
5.4.3	<code>std::memory_order</code>	228
5.4.4	<code>std::atomic_flag</code>	230
5.4.5	<code>std::atomic_ref</code>	231
5.5	STL	232
5.5.1	STL parallel algorithm	232
5.5.2	atomic smart pointer	233
6	Template programming	235
6.1	Template basic	235
6.1.1	c++ template intro	235
6.1.2	View template instantiation	235
6.1.3	template misc	236
6.1.4	explicit template instantiation	237
6.1.5	function & function template	238
6.1.6	template return type	239
6.1.7	class template	240
6.1.8	member template	242
6.1.9	lazy instantiation	243
6.1.10	template & friend	246
6.1.11	variable template, using template	249
6.2	More than basic	250
6.2.1	dependent name	250
6.2.2	template parameter	252
6.2.3	Template type deduction	256
6.2.4	Class template deduction guide	259
6.2.5	object generator	261
6.2.6	<code>std::type_identity</code>	262
6.3	Specialization	263
6.3.1	Template specialization	263
6.3.2	<code>std::conditional</code>	266
6.3.3	Template meta programming	267
6.3.4	Variable template specialization	267
6.4	type_traits	268
6.4.1	Type traits	268
6.4.2	<code>is_pointer</code> & <code>if constexpr</code>	270
6.4.3	<code>int2type</code>	271
6.4.4	<code>std::integral_constant</code>	272
6.4.5	<code>std::is_pointer</code> implementation	274
6.4.6	type modification traits	276
6.4.7	<code>remove_all_pointer</code>	277
6.5	Concept	278
6.5.1	concept	278

6.5.2	requires expression	280
6.5.3	concept example	282
6.5.4	requires clauses	284
6.6	Variadic template	288
6.6.1	variadic template	288
6.6.2	Pack expansion	289
6.6.3	variadic template recursion	292
6.6.4	fold expression	292
6.6.5	variadic template example	294
6.6.6	make tuple	295
6.6.7	make get	297
6.7	CRTP	300
6.7.1	CRTP	300
7	References	305
8	Revision log	305

1 Introduction

본 포스트는 필자가 Modern C++을 공부하면서 중요하다고 생각되는 내용들을 정리한 포스트이다. 각 C++ 표준별로 개선된 사항에 대해 정리하면 다음과 같다. 대부분의 내용은 [1],[2]를 참고하여 작성하였다.

1.1 What is new in modern c++

- **c++11:** 해당 버전 이전을 Legacy c++라고 하고, 이후를 Modern c++이라고 구분할 만큼 많은 변화가 있었다. rvalue reference(&&)와 이동 생성자, 이동 대입 연산자, `auto`, `lambda`, `constexpr` 등 많은 개념이 도입되었다.
 - 그외에도 `nullptr`, `char16_t`, `char32_t` 타입 추가,
 - 중괄호 초기화,
 - 멤버 선언부 초기화,
 - `range-based for`문,
 - `default`, `delete`, `override`, `final` 키워드 추가, 생성자 위임, 생성자 상속,
 - 명시적 형변환,
 - `noexcept`, `decltype` 추가,
 - perfect forwarding,
 - `static_assert()`,
 - 가변 템플릿,
 - 런타임 성능 개선과 컴파일 타임 프로그래밍, 코딩 컨벤션 강화 등 많은 부분이 추가되었다.
- **c++14:** c++11에 추가된 내용을 보강하였다. 리턴 타입 추론으로 c++11의 후행 리턴 개념을 보강하였고 `constexpr` 함수 제약을 완화하여 컴파일 타임 함수 작성 편의성을 향상하였다.
 - 그외에도 `variable template`, `decltype(auto)`, `[[deprecated]]` 키워드,
 - 람다 캡쳐 초기화, 일반화된 람다 표현식 등의 내용이 추가되었다.
- **c++17:** 기존 컴파일러에 의존하던 코드 최적화를 임시 구체화와 복사 생략 보증으로 표준화하였고 `if constexpr`과 클래스 템플릿 인수 추론으로 컴파일 타임 프로그래밍을 강화하였다.
 - 그외에도 `inline variable`, `auto`의 중괄호 초기화 특수 추론 규칙 개선,
 - `enum`의 중괄호 직접 초기화 허용, 람다 캡쳐시 `*this` 이용,
 - `constexpr` 람다 표현식,
 - `static_assert()`의 메세지 생략,
 - `noexcept` 함수 유형 포함,
 - Fold 표현식,
 - 비타입 템플릿 인자에서 `auto` 허용,
 - 16진수 부동 소수점 리터럴, `[[fallthrough]]`, `[[nodiscard]]`, `[[maybe_unused]]` 키워드가 추가되었다.
- **c++20:** `concept`, `requires`가 추가되어 코딩 컨벤션이 강화되었으며 모듈로 컴파일 속도가 향상되었다. 코루틴으로 함수의 일시 정지가 가능해졌으며 삼중 비교 연산자가 추가되어 비교 연산자 구현이 간단해졌다. 축약된 함수 템플릿으로 `auto`를 함수의 인자로 사용할 수 있고, 람다 표현식에서 템플릿 인자 자원으로 람다 표현식의 일반화 프로그래밍이 강화되었다. 그리고 `range-based for`문에서 초기식을 사용할 수 있어 반복문 작성이 좀 더 쉬워졌다.
 - 그외에도 컴파일 타임 프로그래밍 (`constexpr` 함수, `constinit`, `constexpr` 함수)의 추가 제약이 완화되었으며
 - 템플릿 인수 추론 시 `initializer_list`가 개선되었고
 - 람다 표현식에서 파라미터 팩 지원, 캡쳐에서 구조화된 바인딩 지원, 상태없는 람다 표현식의 기본 생성과 복사 대입 지원, 미평가 표현식에서도 람다 표현식 허용,
 - `explicit(bool)` 추가,
 - 인라인 네임스페이스와 단순한 중첩 네임스페이스 결합, 지명 초기화,
 - `new[]`에서 중괄호 집합 초기화로 배열 크기 추론,
 - `using enum` 추가, `__VA_OPT__`, `__has_cpp_attribute()` 매크로 함수 추가, `[[nodiscard]]`의 생성자 지원, `[[nodiscard("reason")]]`, `[[likely]]`, `[[unlikely]]`, `[[no_unique_address]]`가 추가되었다.

1.2 Table summary

Runtime programming	
move (<code>c++11</code>)	(<code>c++11</code>) <ul style="list-style-type: none"> 우측값 참조(<code>&&</code>)와 이동 생성자, 이동 대입 연산자가 추가되어 이동 연산을 지원하며, 임시 객체 대입 시 속도가 향상되었다. 전달 참조가 추가되어 포워딩 함수에서도 효율적으로 함수 인자를 완벽하게 전달할 수 있다.
unrestricted unions (<code>c++11</code>)	(<code>c++11</code>) <ul style="list-style-type: none"> 무제한 공용체(Unrestricted unions)가 추가되어 공용체 멤버에서 생성자/소멸자/가상 함수 사용 제한이 풀렸으며, 메모리 절약을 위한 코딩 자유도가 높아졌다.
Temporary materialization & Copy elision (<code>c++17</code>)	(<code>c++17</code>) <ul style="list-style-type: none"> 임시 구체화(Temporary materialization)와 복사 생략 보증(Copy elision)을 통해 컴파일러의 존적이었던 생성자 호출 및 함수 인수 전달 최적화, 리턴값 최적화 등이 표준화되었다.

Compile time programming	
constexpr (<code>c++11</code>)	(<code>c++11</code>) <ul style="list-style-type: none"> <code>constexpr</code>이 추가되어 컴파일 타임 프로그래밍이 강화되었다. (<code>c++14</code>) <ul style="list-style-type: none"> <code>constexpr</code> 함수의 제약이 완화되어 지역 변수, 2개 이상 리턴문, <code>if</code>, <code>for</code>, <code>while</code> 등을 사용할 수 있게 되었다. (<code>c++17</code>) <ul style="list-style-type: none"> <code>if constexpr</code>이 추가되어 조건에 맞는 부분만 컴파일하고 그렇지 않은 부분은 컴파일에서 제외할 수 있다. (<code>c++20</code>) <ul style="list-style-type: none"> <code>constexpr</code> 함수가 추가되어 컴파일 타임 함수로만 동작할 수 있다. <code>constinit</code>이 추가되어 전역 변수, <code>static</code> 전역 변수, 정적 멤버 변수를 컴파일 타임에 초기화할 수 있다. <code>constexpr</code> 함수 제약 완화가 보강되어 가상 함수, <code>dynamic_cast</code>, <code>typeid()</code>, 초기화되지 않은 지역 변수, <code>try-catch()</code>, 공용체 멤버 변수 활성 전환, <code>asm</code> 등을 사용할 수 있다.
static_assert (<code>c++11</code>)	(<code>c++11</code>) <ul style="list-style-type: none"> <code>static_assert</code>이 추가되어 컴파일 타임 진단이 가능해졌다. (<code>c++17</code>) <ul style="list-style-type: none"> <code>static_assert()</code> 메시지의 생략을 지원한다.

advanced template (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> 가변 템플릿 파라미터 팩이 추가되어 가변 인자 (...)와 같이 갯수와 타입이 정해지지 않은 템플릿 인자를 사용할 수 있다. <code>sizeof...</code>() 연산자가 추가되어 가변 템플릿에서 파라미터 팩의 인자수를 구할 수 있다. <code>extern</code>으로 템플릿을 선언할 수 있으며 템플릿 인스턴스 중복 생성을 없앨 수 있다. 템플릿 오른쪽 꺽쇠 괄호 파싱을 개선하여 템플릿 인스턴스화 시 >가 중첩되어 >>와 같이 되더라도 공백을 추가할 필요가 없어졌다. <p>(c++14)</p> <ul style="list-style-type: none"> <code>variable template</code>이 추가되어 변수도 템플릿으로 만들 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> 클래스 템플릿 인수 추론이 추가되어 함수 템플릿처럼 템플릿 인스턴스화시 타입을 생략할 수 있다. 클래스 템플릿 인수 추론 사용자 정의 가이드가 추가되어 클래스 템플릿 인수 추론 시 컴파일러에게 가이드를 줄 수 있다. 비타입 템플릿 인자에서 <code>auto</code>를 허용한다. Fold 표현식이 추가되어 가변 템플릿에서 파라미터 팩을 재귀적으로 반복하여 전개할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> 축약된 함수 템플릿이 추가되어 <code>auto</code> 타입을 사용할 수 있다. 사실 상 함수 템플릿의 간략한 표현이다. 비타입 템플릿 인자 규칙이 완화되어 실수 타입과 리터럴 타입을 사용할 수 있다. 클래스 템플릿 인수 추론 시 <code>initializer_list</code>인 경우가 개선되어 ,2,3<code>std::vector v1</code>처럼 템플릿 인자를 명시하지 않아도 된다.
concept (c++20)	<p>(c++20)</p> <ul style="list-style-type: none"> <code>concept</code>과 <code>requires</code>가 추가되어 템플릿 인자나 <code>auto</code>에 제약 조건을 줄 수 있다.

Coding convention enhancement

advanced type and literals ([c++11](#))

([c++11](#))

- 타입 카테고리를 수립하여 컴파일 타임 프로그래밍이나 템플릿 메타 프로그램이 시 코딩 컨벤션을 강화할 수 있다.
- `using`을 이용한 타입 별칭이 추가되어 `typedef`보다 좀 더 직관적인 표현이 가능해졌다.
- `nullptr` 리터럴이 추가되어 포인터에 안전한 코딩 컨벤션이 가능해졌다.
- `long long` 타입이 추가되어 최소 8byte 크기를 보장한다.
- `ll, ull, LL, ULL` 리터럴이 추가되어 `long long`용 정수형 상수를 제공한다.
- `char16_t, char32_t` 타입이 추가되어 UTF-16 인코딩, UTF-32 인코딩을 모두 지원한다.
- `u8"", u"", U"", u"(char), U"(char)` 리터럴이 추가되어 유니코드를 지원하는 `char16_t, char32_t` 타입용 문자 상수를 제공한다.
- `R"()"` 리터럴이 추가되어 개행이나 이스케이프 문자를 좀 더 편하게 입력할 수 있다.

([c++14](#))

- 이진 리터럴이 추가되어 `0b, 0B` 접두어로 이진수 상수를 표현할 수 있다.
- 숫자 구분자가 추가되어 `1'000'000`과 같이 작은 따옴표 '`'`를 숫자 사이에 선택적으로 넣을 수 있어 가독성이 좋아졌다.

([c++17](#))

- 16진수 부동 소수점 리터럴이 추가되어 `0xA, 9p11`과 같이 16진수로 실수를 표현할 수 있다.
- `u8"(char)` 리터럴이 추가되어 유니코드를 지원하는 1byte 크기의 문자 상수를 지원한다.

([c++20](#))

- `char8_t` 타입이 추가되어 UTF-8 인코딩 문자를 지원한다.
- 정수에서 2의 보수 범위를 보장한다.
- 사용자 정의 리터럴 인자 규칙에 `char8_t`가 추가되었다.

noexcept ([c++11](#))

([c++11](#))

- `noexcept`가 추가되어 함수의 예외 방출 여부를 보증하며 소멸자는 기본적으로 `noexcept`로 작동한다.
- `noexcept` 연산자가 추가되어 해당 함수가 `noexcept`인지 컴파일 타임에 검사할 수 있다.

([c++17](#))

- `noexcept`가 함수 유형에 포함되어 예외 처리에 대한 코딩 컨벤션을 좀 더 단단하게 할 수 있다.

explicit type conversion (c++11)	(c++11) <ul style="list-style-type: none"> • <code>explicit</code> 형변환 연산자가 추가되어 명시적으로 형변환할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> • <code>explicit(bool)</code>이 추가되어 특정 조건일 때만 <code>explicit</code>으로 동작하게 할 수 있다.
attribute (c++11)	(c++11) <ul style="list-style-type: none"> • <code>attribute</code>가 추가되어 컴파일러에게 부가 정보를 전달하는 방식을 표준화하였다. <p>(c++14)</p> <ul style="list-style-type: none"> • <code>[[deprecated]]</code>가 추가되어 소멸 예정인 것을 컴파일 경고로 알려준다. <p>(c++17)</p> <ul style="list-style-type: none"> • <code>[[fallthrough]]</code>가 추가되어 <code>switch()</code>에서 의도적으로 <code>break</code>를 생략하여 다음 case로 제어를 이동시킬 때 발생하는 컴파일 경고를 차단할 수 있다. • <code>[[nodiscard]]</code>가 추가되어 리턴값을 무시하지 않도록 컴파일 경고를 해준다. • <code>[[maybe_unused]]</code>가 추가되어 사용되지 않은 객체의 컴파일 경고를 차단할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> • <code>[[nodiscard]]</code>의 생성자 지원, <code>[[nodiscard("reason")]]</code>이 추가되었다. • <code>[[likely]], [[unlikely]]</code>가 추가되어 컴파일러에게 최적화 힌트를 줄 수 있다. • <code>[[no_unique_address]]</code>가 추가되어 아무 멤버 변수가 없는 객체의 크기를 최적화할 수 있다.

Coding convenience enhancement	
advanced namespace (c++11)	(c++11) <ul style="list-style-type: none"> • 인라인 네임스페이스가 추가되어 API 버전 구성이 편리해졌다. <p>(c++14)</p> <ul style="list-style-type: none"> • 단순한 중첩 네임스페이스가 추가되어 ::로 표현할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> • 인라인 네임스페이스와 단순한 중첩 네임스페이스를 결합하여 표현할 수 있다.

advanced initialization ([c++11](#))

([c++11](#))

- 중괄호 초기화를 제공하여 클래스, 배열, 구조체 구분없이 중괄호로 일관성있게 초기화를 할 수 있으며 초기화 패싱 오류를 해결하였다.
- 중괄호 복사 초기화로 함수 인수 전달, 리턴문 작성을 간결화할 수 있다.
- 중괄호 초기화시 인자의 암시적 형변환을 일부 차단하여 코딩 컨벤션이 개선되었다.
- `initializer_list`가 추가되어 `vector` 등 컨테이너의 초기 요소 추가가 간편해졌다.
- 멤버 선언부 초기화가 추가되어 non-static 멤버 변수의 초기화가 쉬어졌다.

([c++14](#))

- `non-static` 멤버 선언부 초기화 시 집합 초기화를 허용한다.

([c++20](#))

- 지명 초기화가 중괄호 집합 초기화 시 변수명을 지명하여 값을 초기화할 수 있다.
- 비트 필드 선언부 초기화가 추가되었다.
- `new[]`에서 중괄호 집합 초기화로 배열 크기 추론이 추가되어 배열 크기를 명시하지 않아도 된다.

advanced control statement ([c++11](#))

([c++11](#))

- `range-based for()`가 추가되어 컨테이너 요소의 탐색 처리가 쉬워졌다.

([c++17](#))

- 초기식을 포함하는 `if()`, `switch()`가 추가되어 함수 리턴 값을 평가하고 소멸하는 코드가 단순해졌다.

([c++20](#))

- `range-based for()`에서 초기식이 추가되었다.

advanced class (c++11)	(c++11) <ul style="list-style-type: none"> • <code>default</code>, <code>delete</code>가 추가되어 암시적으로 생성되는 멤버 함수의 사용 여부를 좀 더 명시적으로 정의할 수 있다. • <code>override</code>가 추가되어 가상 함수 오버라이딩 코딩 규약이 좀 더 단단해졌다. • <code>final</code>이 추가되어 가상 함수를 더 이상 오버라이딩 못하게 할 수 있고 강제적으로 상속을 제한할 수 있다. • 생성자 위임이 추가되어 생성자의 초기화 리스트 코드가 좀 더 간결해졌다. • 생성자 상속이 추가되어 부모 객체의 생성자도 상속받아 사용할 수 있어 자식 객체의 생성자 재정의 코드가 좀 더 간결해졌다. • 멤버 함수 참조 지정자가 추가되어 멤버 함수에 <code>&</code>, <code>&&</code>로 lvalue로 호출될 때와 rvalue로 호출 될 때를 구분하여 오버로딩을 할 수 있다.
auto decltype trailing return type (c++11)	(c++11) <ul style="list-style-type: none"> • <code>auto</code>와 <code>decltype()</code>이 추가되어 값으로부터 타입을 추론하여 코딩이 간편해졌다. 후행 리턴(trailing return)이 추가되어 함수 인자에 의존하여 리턴 타입을 결정하며 좀 더 동적인 함수 설계가 가능해졌다. <p>(c++14)</p> <ul style="list-style-type: none"> • <code>decltype(auto)</code>가 추가되어 <code>decltype()</code>의 () 내 표현식이 복잡할 경우 좀 더 간결하게 작성할 수 있다. • 리턴 타입 추론이 추가되어 후행 리턴 대신 <code>auto</code>나 <code>decltype(auto)</code>를 사용할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> • <code>auto</code>의 중괄호 초기화 특수 추론 규칙이 개선되어 <code>auto a1; </code> 시 <code>initializer_list</code>가 아니라 <code>int</code>로 추론된다.
scoped enum (c++11)	(c++11) <ul style="list-style-type: none"> • 범위 있는 열거형(scoped enum)이 추가되어 이름 충돌 회피가 쉬워졌고 암시적 형변환을 차단하며 전방 선언도 지원한다. <p>(c++17)</p> <ul style="list-style-type: none"> • 열거형의 중괄호 직접 초기화를 허용하여 암시적 형변환을 차단하는 사용자 정의 열거형의 사용이 좀 더 쉬워졌다. <p>(c++20)</p> <ul style="list-style-type: none"> • <code>using enum</code>이 추가되어 열거형의 이름 없이 열거자를 유효 범위 내에서 사용할 수 있다.

lambda expression, closure (c++11)	<p>(c++11)</p> <ul style="list-style-type: none"> 람다 표현식이 추가되어 1회용 익명 함수를 만들 수 있다. <p>(c++14)</p> <ul style="list-style-type: none"> 람다 캡쳐 초기화가 추가되어 람다 표현식 내에서 사용하는 임의 변수를 정의하여 사용할 수 있다. 일반화된 람다 표현식이 추가되어 <code>auto</code>를 받아 마치 함수 템플릿처럼 사용할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> 람다 캡쳐 시 <code>*this</code>가 추가되어 객체 자체를 복제하여 사용할 수 있다. <code>constexpr</code> 람다 표현식이 추가되어 람다 표현식도 컴파일 타임 함수로 만들 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> 람다 표현식에서 템플릿 인자를 지원한다. 람다 캡쳐에서 파라미터 팩을 지원한다. 람다 캡쳐에서 구조화된 바인딩을 지원한다. 상태없는 람다 표현식의 기본 생성과 복사 대입을 지원한다. 미평가 표현식에서도 람다 표현식을 허용하기 때문에 <code>decltype()</code> 안에서 사용할 수 있다.
advanced variables (c++17)	<p>(c++17)</p> <ul style="list-style-type: none"> 인라인 변수가 추가되어 헤더 파일에 정의된 변수를 여러 개의 cpp에서 <code>#include</code> 하더라도 중복 정의없이 사용할 수 있다. 또한 클래스 정적 멤버 변수를 선언부에서 초기화할 수 있다.
structured bindings (c++17)	<p>(c++17)</p> <ul style="list-style-type: none"> 구조화된 바인딩(structured bindings)이 추가되어 배열, <code>pair</code>, <code>tuple</code>, <code>class</code> 등 내부 요소나 멤버 변수에 쉽게 접근할 수 있다.
advanced operator (c++20)	<p>(c++20)</p> <ul style="list-style-type: none"> 삼중 비교 연산자가 추가되어 비교 연산자 구현이 간소화되었다. 삼중 비교 연산자를 <code>default</code>로 정의할 수 있다. 비트 쉬프트 연산자의 기본 비트가 표준화되어 « 1은 곱하기 2의 효과가 있는 비트(즉, 0)으로 채워지고 » 1은 나누기 2의 효과가 있는 비트 (즉, 양수면 0, 음수면 1)로 채워진다.

module (c++20)	(c++20) <ul style="list-style-type: none"> 모듈이 추가되어 전처리 사용 방식을 개선하여 컴파일 속도를 향상시키고, #include 순서에 따른 종속성 문제, 선언과 정의 분리 구성의 불편함, 기호 충돌 문제를 해결하였다.
coroutine (c++20)	(c++20) <ul style="list-style-type: none"> 코루틴이 추가되어 함수의 일시 정지 후 재개가 가능해졌다.
misc	(c++11) <ul style="list-style-type: none"> alignas(), alignof()가 추가되어 메모리 정렬 방식을 표준화하였다. 가변 매크로가 추가되어 c언어와 호환성이 높아졌다. 멤버의 sizeof() 시 동작이 개선되어 객체를 인스턴스화 하지 않아도 객체 멤버의 크기를 구할 수 있다. <p>(c++17)</p> <ul style="list-style-type: none"> __has_include가 추가되어 #include 하기 전에 파일이 존재하는지 확인할 수 있다. <p>(c++20)</p> <ul style="list-style-type: none"> __VA_OPT__가 추가되어 가변 인자가 있을 경우에는 괄호 안의 값으로 치환하고 없을 경우에는 그냥 비워둔다. __has_cpp_attribute() 매크로 함수가 추가되어 c++11부터 추가된 attribute가 지원되는지 확인할 수 있다. 언어 지원 테스트 매크로가 추가되어 c++11부터 추가된 언어 기능을 지원하는지 테스트할 수 있다.

deprecated & removed

(c++11)

- 동적 예외 사양은 `deprecated`되었다. 예외를 나열하는 것 보다 `noexcept`로 예외를 방출하느냐 안하느냐만 관심을 둔다.
- `export` 템플릿은 제대로 구현한 컴파일러는 드물고 세부 사항에 대한 의견이 일치하지 않아 c++11부터 완전히 remove 되었다.

(c++17)

- 동적 예외 사양 관련해서 `throw()`가 `deprecated`되었다. 이제 `noexcept`만 사용해야 한다.
- `&&`, `&=`등이 특수기호가 없는 인코딩을 사용하는 곳을 위해 제공했던 trigraph가 remove되었다.
- 변수를 CPU 레지스터에 배치하도록 힌트를 주는 `register` 가 `deprecated`되었다.
- `bool`의 증감 연산이 `deprecated`되었다.

(c++20)

- 람다 캡쳐에서 `[=]` 사용 시 `this`의 암시적 캡쳐가 `deprecated`되었으므로 명시적으로 작성해야 한다.
- `volatile` 일부가 `deprecated`되었다.

2 Intermediate

2.1 Temporary

```
1 class Point{
2     int x,y;
3     public:
4
5     Point(int x, int y) : x(x), y(y) { std::cout << "Point(int,int)" << std::endl; }
6     ~Point() { std::cout<<"~Point()" << std::endl; }
7 };
8
9 Point pt(1,2); // --> 일반 객체
10 Point (1,2); // --> 임시 객체
11
12 pt.x = 10;           // ok
13 Point(1,2).x = 10; // error (rvalue라고 부름)
14 Point(1,2).set(10,20) // ok (멤버함수는 불러와지므로 상수는 아님)
15
16 Point *p1 = &pt;      // ok
17 Point *p2 = &Point(1,2) // error (임시객체는 포인터로 가르킬 수 없다)
18
19 Point& r1 = pt;      // ok
20 Point& r2 = Point(1,2); // error(임시객체는 참조로 가르킬 수 없다)
21 Point&& r3 = Point(1,2); // ok (rvalue reference 문법)
22
23 const Point &r4 = Point(1,2) // ok(상수 참조는 가능, 일반 객체로 승격되는 효과)
```

객체를 함수 인자로만 사용한다면 임시객체로 전달하는게 효율적일 수 있다. (`const` reference로 받아야만 함)

```
1 void foo(const Point& pt) { std::cout << "foo" << std::endl; }
2
3 int main() {
4     Point pt(1,2);
5     foo(pt);          // pt는 foo에 넣기 위해서만 생성한 객체이므로 함수가 불린 후 바로 파괴되는게 좋다.
6
7     foo(Point(1,2)) // 임시 객체를 활용하자
8     foo( {1, 2} )   // 이렇게 전달하는 것도 가능하다. 컴파일러가 Point1,2로 바꿔줌
9     std::cout<<"-----"<<std::endl;
10 }
```

```
1 void foo(const std::string& s) { }
2 void goo(std::string_view s) { } // call-by-value임에 유의!
3
4 int main() {
5     foo("Practice make perfect"); // 컴파일러에서 string("Practice make perfect")로 변환해줌
6
7     goo("Practice make perfect"); // string_view는 문자열의 복사본을 생성하지 않고 이미 상수 메모리의
8     존재하는 문자열을 가르킨다.
9 }
```

2.1.1 return-by-value vs return-by-reference

```
1 Point pt(1,2);
2 Point f3() { return pt; }
3 Point& f4() { return pt; }
4
5 Point& f5() {
6     Point pt(1,2);
7     return pt;    // error. 지역객체를 return-by-reference하면 안됨!
8 }
9
10 int main() {
```

```
11     f3().x = 10; // error. (return-by-value는 복사본이 생성되어 임시객체이므로 rvalue임)
12     f4().x = 10; // ok. pt.x = 10
13 }
```

```
1 class Counter {
2     int count{0};
3
4     Counter& increment() { //return-by-reference로 하는걸 잊으면 안됨!
5         ++count;
6         return *this;
7     }
8     int get() const { return count; }
9 }
10
11 int main() {
12     Counter c;
13     c.increment().increment().increment();
14     std::cout << c.get() << std::endl;
15 }
```

2.1.2 temporary and casting

```
1 struct Base {
2     int value = 10;
3
4     Base() = default;
5     Base(const Base&b) : value(b.value)
6     { std::cout << "copy constructor" << std::endl; }
7 }
8
9 struct Derived : public Base {
10     int value = 10;
11 }
12
13 int main() {
14     Derived d;
15
16     std::cout << d.value << std::endl; // 20
17     std::cout << static_cast<Base&>(d).value << std::endl; // 10 good. 참조 캐스팅을 통해 d를 Base로
18     생각하게 하여 값을 가져온다.
19     std::cout << static_cast<Base>(d).value << std::endl; // 10 no. Base 클래스의 복사본이 생성되며
20     거기서 값을 가져온다.
21
22     static_cast<Base&>(d).value = 100;
23     static_cast<Base>(d).value = 100; // error. 캐스팅은 항상 reference로 하자!
}
```

2.2 Conversion

이번 섹션에서는 객체 변환에 대한 다양한 문법과 기법에 대해 설명한다.

- 변환 연산자, 변환 생성자, `explicit` 생성자의 개념
- safe bool 개념과 `explicit` 변환 연산자에 대해 살펴본다.
- `nullptr`과 return type resolver 기술
- temporary proxy 기술
- lambda expression과 함수 포인터 변환

2.2.1 Conversion constructor, conversion operator

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5 };
6
7 int main() {
8     int pn;          // primitive type
9     Int32 un;      // user type 값을 지정해주지 않아도 쓰레기값이 아닌 0으로 초기화된다
10
11    pn = un;
12 }
```

int를 대체하기 위한 Int32 클래스에 대해 알아보자. 새로 정의한 타입은 기존에 타입과도 호환이 되는게 좋기 때문에 pn = un과 같이 서로 변환이 가능해야 한다.

pn = un이 호출된 순간 컴파일러는 un.operator int() 함수를 찾게 되는데 이러한 연산자를 **변환 연산자 (conversion operator)**라고 한다.

```
1 operator TYPE() { 변환 연산자
2     return value;
3 }
```

변환 연산자는 반환 타입이 함수 이름에 포함되어 있으므로 반환 타입을 표기하지 않는다.

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5     operator int() const { return value; } const를 추가하여 상수 객체 또한 동작하도록 해준다
6 };
7
8 int main() {
9     int pn;
10    Int32 un;
11    const Int32 un2;
12
13    pn = un;
14    pn = un2;
15 }
```

다음으로 un = pn과 같이 반대의 경우를 보자. 이런 경우 컴파일러는 un.operator=(int) 대입연산자가 존재하는지 먼저 검사한다. 만약 없다면 Int32(int)와 같이 default 대입 연산자가 있는지 검사한다.

```
1 class Int32 {
2     int value;
3     public:
4     Int32() : value(0) {}
5     Int32(int n) : value(n) {} 변환 생성자
6     operator int() const { return value; }
7 };
8
9 int main() {
10    int pn;          // primitive type
11    Int32 un;      // user type 값을 지정해주지 않아도 쓰레기값이 아닌 0으로 초기화된다
12    const Int32 un2;
13
14    pn = un;
15    pn = un2;
16    un = pn; // 변환 생성자 호출!
17 }
```

변환 생성자에 대해 좀 더 자세히 알아보자

```

1  class Int32 {
2      int value;
3  public:
4      Int32(int n) : value(n) {}
5  };
6
7  int main() {
8      Int32 n1(3);      // 1. direct initialization
9      Int32 n2 = 3;    // 2. copy initialization
10     Int32 n3{3};    // 3. direct init. (c++11)
11     Int32 n4 = {3}; // 4. copy init. (c++11)
12
13     n1 = 3; // conversion (int -> Int32)
14 }
```

변환 생성자가 있는 경우 위와 같이 변환 이외에도 4가지 초기화 코드를 만들 수 있다. 이 중 2번에 대해 자세히 알아보자.

`Int32 n2 = 3`이 호출되면 컴파일러는 이를 `Int32 n2 = Int32(3)`으로 변환한다. 이는 임시 객체(temporary)이므로 **c++98** 시절까지는 복사 생성자를 통해 `n2`에 대입되지만 **c++11** 이후부터는 `move` 생성자를 통해 `n2`로 대입된다. 하지만 대부분의 컴파일러에서 최적화 옵션을 켜면 임시 객체 생성이 제거된다.

```

1  class Int32 {
2      int value;
3  public:
4      Int32(int n) : value(n) {}
5      Int32(const Int32&) = delete; // 복사 생성자 제거
6  };
7
8  int main() {
9      Int32 n1(3);
10     Int32 n2 = 3; // error! c++14까지는 에러가 발생하지만 c++17부터는 문법적으로 복사 생성을 하지 않기
11     때문에 ok
12     Int32 n3{3};
13     Int32 n4 = {3};
14
15     n1 = 3;
16 }
```

다음으로 마지막 줄의 `n1 = 3`을 살펴보자. 이는 컴파일러에 의해 `n1 = Int32(3)`으로 변환된다. 즉, 임시 객체가 생성되고 디폴트 대입 연산자를 사용해서 `n1`에 대입되는 형태이다. 대부분의 컴파일러가 최적화를 통해 임시 객체 생성이 제거되지만 **디폴트 대입 연산자는 반드시 존재해야 한다.**

```

1  class Int32 {
2      int value;
3  public:
4      Int32(int n) : value(n) {}
5      Int32(const Int32&) = delete; // 복사 생성자 제거
6      Int32& operator=(const Int32&) = delete; // 디폴트 대입 연산자 제거
7  };
8
9  int main() {
10     Int32 n1(3);
11     //Int32 n2 = 3;
12     Int32 n3{3};
13     Int32 n4 = {3};
14
15     n1 = 3; // error. 디폴트 대입 연산자가 없으면 모든 버전에 대해 에러가 발생한다.
16 }
```

2.2.2 explicit constructor

```

1  class Vector{
2     public:
3     Vector(int size) {}
```

```

4   };
5   void foo(Vector v) {}
6   int main() {
7     Vector v1(3);
8     Vector v2 = 3;
9     Vector v1{3};
10    Vector v1 = {3};

11
12    v1 = 3; // 논리적으로 벡터에 3을 넣는게 맞지 않음!
13    foo(3); // ok but no. Vector v = 3의 형태로 복사생성자가 호출됨
14 }

```

explicit 생성자를 사용하면 생성자가 암시적 변환의 용도로 사용될 수 없게 한다. 이런 경우 직접 초기화(direct initialization)만 가능하고 복사 초기화(copy initialization)은 사용할 수 없다.

```

1 class Vector{
2   public:
3     explicit Vector(int size) {}
4   };
5   void foo(Vector v) {}
6   int main() {
7     Vector v1(3);
8     Vector v2 = 3; // error
9     Vector v1{3};
10    Vector v1 = {3}; // error

11
12    v1 = 3; // error
13    foo(3); // error
14 }

```

`explicit`을 사용할 때는 클래스에 따라 `explicit`을 사용할 지 잘 판단해야 한다.

```

1 void f1(Int32 n) {}
2 void f2(Vector v) {}

3
4 f1(3); // ok. 논리적으로 맞으므로 Int32 생성자에는 explicit를 붙일 필요 없음
5 f2(3); // ok but no. 논리적으로 맞지 않으므로 Vector 생성자에는 explicit를 붙여야함

```

2.2.3 explicit conversion operator

객체의 유효성을 `if`문을 통해 비교하고 싶다고 하자.

```

1 class Machine {
2   int data = 10;
3   bool state = true;
4 };
5 int main() {
6   Machine m;
7   if(m) {} // 객체의 유효성을 if문을 통해 비교하고자 한다
8 }

```

컴파일을 해보면 'Machine을 `bool`로 변환할 수 없다'는 에러 구문이 발생하는데 이는 곧 `Machine을 bool로만 변화할 수 있으면 if문이 동작한다는 얘기와 같다.`

```

1 class Machine {
2   int data = 10;
3   bool state = true;
4   public:
5     operator bool() { return state; } // bool 연산자
6   };
7
8   int main() {
9     Machine m;
10    if(m) {} // ok
11 }

```

`if(m)` 구문은 이제 정상적으로 작동하지만 `operator bool()`은 side effect가 많아서 사용에 주의해야 한다. 예를 들어 다음과 같은 side effect가 발생한다.

```
1 class Machine {
2     int data = 10;
3     bool state = true;
4     public:
5         operator bool() { return state; }
6     };
7
8     int main() {
9         Machine m;
10
11     bool b1 = m; // ok
12     bool b2 = static_cast<bool>(m); // ok
13     m << 10; // 문법이 이상하지만 컴파일 된다. 즉, 버그의 원인이 될 수 있다.
14     if(m) { ... }
15 }
```

이러한 문제를 해결하기 위해 c++11부터 생성자 뿐만 아니라 변환 연산자도 `explicit`을 붙일 수 있다. 이러한 기술을 **safe bool**이라고 부른다.

```
1 class Machine {
2     int data = 10;
3     bool state = true;
4     public:
5         explicit operator bool() { return state; }
6     };
7
8     int main() {
9         Machine m;
10
11     bool b1 = m; // error
12     bool b2 = static_cast<bool>(m); // ok. 명시적 변환이므로
13     m << 10; // error
14     if(m) { ... }
15 }
```

c++98부터 `explicit` 변환 생성자 구문을 제공하였으나 c++11이 되면서 `explicit` 변환 연산자 문법이 추가되었다. 그리고 c++20에는 `explicit(bool)` 문법이 제공되었다. `explicit(bool)` 문법에 대해 알아보자.

2.2.4 `explicit(bool)`, c++20

```
1 template<class T>
2 class Number {
3     T value;
4     public:
5         explicit(true) Number(T v) : value(v) {}
6     };
7
8     int main() {
9         Number n1 = 10; // error
10        Number n2 = 3.4; // error. explicit이 true이므로 에러 발생. 만약 explicit(false)이면 컴파일이
11        정상적으로 된다
11 }
```

위 코드에서 `explicit(true)`를 하면 `expliciit` 키워드를 사용한다는 의미이고 반대로 `explicit(false)`를 하면 사용하지 않는다는 의미이다. 만약 정수형일 때만 `explicit`을 사용하지 않고 `float`일 때는 `explicit`을 사용하고 싶으면 어떻게 해야 할까?

```
1 template<class T>
2 class Number {
3     T value;
4     public:
5         explicit(!std::is_integral_v<T>)
```

```

6     Number(T v): value(v) {}
7 };
8 int main() {
9     Number n1 = 10; // ok. explicit(false)이므로
10    Number n2 = 3.4; // error. explicit(true)이므로
11 }

```

2.2.5 Conversion example: nullptr

`nullptr`은 포인터 초기화 시 0 대신 사용하는 C++ 키워드이다. 원래 boost 라이브러리에 있는 도구를 C++11을 만들면서 표준에 추가되었다. 이번 섹션에서는 boost에 있는 `nullptr`을 구현하면서 `nullptr`의 개념에 대해 다시 한번 숙지해보자.

```

1 void foo(int* p) {}
2 void goo(char* p) {}

3
4 struct nullptr_t {
5     template<class T>
6     constexpr operator T*() const { return 0; }
7 };
8 nullptr_t xnullptr;

9
10 int main() {
11     foo(xnullptr);
12     goo(xnullptr); // nullptr은 이미 키워드이므로 임의의 xnullptr을 정의하였다.
13 }

```

위 코드 예제와 유사하게 실제 `nullptr`의 타입도 `std::nullptr_t`이다.

2.2.6 Conversion example: return type resolver

Return type resolver는 좌변을 보고 우변의 반환 타입을 자동으로 결정하는 기술을 말한다. 다음과 같은 예제 코드를 보자.

```

1 template<class T>
2 T* Alloc(std::size_t sz) {
3     return new T[sz];
4 }
5
6 int main() {
7     int* p1 = Alloc<int>(10);
8     double* p2 = Alloc<double>(10); // <double>과 같은 꺽쇠를 사용하지 않고 바로 Alloc을 사용할 수는
9     없을까?

```

`Alloc(10)`으로 바로 받기 위해서는 `Alloc`이 함수가 아니라 구조체여야 한다.

```

1 struct Alloc {
2     std::size_t size;
3     Alloc(std::size_t sz) : size(sz) {}
4
5     template<class T>
6     operator T*() { return new T[size]; }
7 };
8
9 int main() {
10    int* p1 = Alloc(10);
11    double* p2 = Alloc(10); // 변환연산자에 의해 자동으로 타입이 결정되어 할당된다.

```

2.2.7 Lambda expression and conversion

일반적으로 람다 표현식은 `auto` 변수에 담아서 사용하지만 함수 포인터로도 람다 표현식을 담을 수 있다.

```

1 int main() {

```

```

2     auto f1 = [](int a, int b) { return a + b; }
3     int (*f2)(int, int) = [](int a, int b) { return a + b; } // 람다 표현식은 임시객체를 만들어주므로
4         임시객체.operator 함수포인터()가 호출되어 함수포인터에 할당된다.
}

```

2.3 Constructor

이번 섹션에서는 생성자의 호출 원리에 대해 살펴본다. 다음과 같은 4개의 구조체를 보자.

```

1 struct BM {
2     BM() { cout << "BM()" << endl; }
3     ~BM() { cout << "~BM()" << endl; }
4 };
5 struct DM {
6     DM() { cout << "DM()" << endl; }
7     DM(int) { cout << "DM(int)" << endl; }
8     ~DM() { cout << "~DM()" << endl; }
9 };
10 struct Base {
11     BM bm;
12     Base() { cout << "Base()" << endl; }
13     Base(int a) { cout << "Base(int)" << endl; }
14     ~Base() { cout << "~Base()" << endl; }
15 };
16 struct Derived : public Base {
17     DM dm;
18     Derived() { cout << "Derived()" << endl; }
19     Derived(int a) { cout << "Derived(int)" << endl; }
20     ~Derived() { cout << "~Derived()" << endl; }
21 };
22
23 int main() {
24     Derived d1; // call Derived::Derived()
25     Derived d2(7); // call Derived::Derived(int)
26 }

```

d1, d2을 호출하는 순간 컴파일러에 의해 자동 생성된 코드가 실행된다.

```

1 struct Base {
2     BM bm;
3     Base() : bm() { cout << "Base()" << endl; }
4     Base(int a) : bm() { cout << "Base(int)" << endl; }
5     ~Base() { cout << "~Base()" << endl; bm. BM(); }
6 };
7 struct Derived : public Base {
8     DM dm;
9     Derived() : Base(), dm() { cout << "Derived()" << endl; }
10    Derived(int a) : Base(), dm() { cout << "Derived(int)" << endl; }
11    ~Derived() { cout << "~Derived()" << endl; dm. DM(); Base(); }
12 };
13
14 int main() {
15     Derived d1; // call Derived::Derived()
16 }

```

생성자, 소멸자 호출자를 정확히 명시해주지 않아도 컴파일러가 **기반 클래스 및 멤버 데이터의 생성자(소멸자)**를 호출해주는 코드를 생성해준다.

다음으로 호출 순서를 정확히 아는 것이 중요하다. d1을 호출하면 **bm() → Base() → dm() → Derived()**가 순서대로 호출된다. 임의로 코드의 위치를 바꿔도 **사용자가 순서대로 호출 순서를 변경할 수 없다.**

또한, 컴파일러가 생성한 코드는 항상 디폴트 생성자를 호출한다. **기반 클래스나 멤버 데이터에 디폴트 생성자가 없는 경우 반드시 사용자가 디폴트가 아닌 다른 생성자를 호출하는 코드를 작성해야 한다.**

```

1 struct BM {
2     BM() { cout << "BM()" << endl; }
3     ~BM() { cout << "~BM()" << endl; }

```

```

4   };
5   struct DM {
6     //DM() { cout << "DM()" << endl; }
7     DM(int) { cout << "DM(int)" << endl; }
8     ~DM() { cout << "~DM()" << endl; }
9   };
10  struct Base {
11    BM bm;
12    //Base() { cout << "Base()" << endl; }
13    Base(int a) { cout << "Base(int)" << endl; }
14    ~Base() { cout << "~Base()" << endl; }
15  };
16  struct Derived : public Base {
17    DM dm;
18    Derived() : Base(0), dm(0) { cout << "Derived()" << endl; } // Base(0), dm(0)을 호출하지 않으면 에러
19    발생!
20    Derived(int a) : Base(0), dm(0) { cout << "Derived(int)" << endl; }
21    ~Derived() { cout << "~Derived()" << endl; }
22  };
23
24  int main() {
25    Derived d1;
26    Derived d2(7);
27 }

```

2.3.1 Base from member idioms

다음과 같이 버퍼를 사용하는 스트림 예제 코드를 보자.

```

1  class Buffer {
2    public:
3      Buffer(std::size_t sz) { cout << "initialize buffer" << endl; }
4      void use() { cout << "use buffer" << endl; }
5  };
6
7  class Stream {
8    public:
9      Stream(Buffer& buf) { buf.size(); }
10 };
11
12 int main() {
13   Buffer buf(1024);
14   Stream s(buf);
15 }

```

위 코드는 정상적으로 initialize buffer를 통해 버퍼를 초기화한 후 use buffer가 호출되어 버퍼를 사용한다.
다음으로 buffer를 멤버함수로 두고 싶은 경우를 생각해보자.

```

1  class Buffer {
2    public:
3      Buffer(std::size_t sz) { cout << "initialize buffer" << endl; }
4      void use() { cout << "use buffer" << endl; }
5  };
6  class Stream {
7    public:
8      Stream(Buffer& buf) { buf.size(); }
9  };
10
11 class StreamWithBuffer : public Stream {
12   Buffer buf(1024);
13   public:
14     StreamWithBuffer() : Stream(buf) {}
15 };
16
17 int main() {

```

```
18     StreamWithBuffer swf; // error. use buffer, initialize buffer 순으로 잘못 호출됨!
19 }
```

위 코드는 초기화되지 않은 버퍼를 사용하는 문제가 발생한다. 즉, **멤버 데이터보다 기반 클래스의 생성자가 먼저 호출되는 문제가 발생한다.**

```
1 class StreamWithBuffer : public Stream {
2     Buffer buf(1024);
3     public:
4     StreamWithBuffer() : Stream(buf), buf(1024) {} // 컴파일러에 의해 buf(1024)가 늦게 호출된다
5 };
```

이를 해결하는 방법은 다중 상속으로 해결해야 한다.

```
1 class StreamBuffer {
2     protected:
3     Buffer buf(1024);
4 };
5
6 class StreamWithBuffer : public StreamBuffer, public Stream { // 다중 상속을 통해 문제를 해결
7     public:
8     StreamWithBuffer() : StreamBuffer(), Stream(buf) {}
9 };
```

위 기술은 c++ 초기 설계 당시 ostream을 만들 때 사용한 방법이며 "**Base from member(c++ idioms)**"라는 이름을 가지고 있다.

2.3.2 Constructor and virtual function

가상함수를 호출하는 시점에 따라 Dervied 클래스의 함수를 호출하거나 Base의 함수를 호출하는 동작이 달라진다.

```
1 class Base{
2     public:
3     Base() { vfunc(); }    // Base vfunc
4     void foo() { vfunc(); } // Derived vfunc
5     virtual void vfunc() { cout << "Base vfunc()" << endl; }
6 };
7
8 class Derived : public Base {
9     int data{10};
10    public:
11    virtual void vfunc() override{ cout << "Derived vfunc()" << data << endl; }
12 };
13
14 int main() {
15     Derived d;
16     d.foo(); // Derived의 vfunc가 실행된다.
17 }
```

생성자에서는 가상 함수가 동작하지 않는다. 왜 이렇게 설계되었을까? Dervied 생성자는 컴파일러를 통해 다음과 같이 생성된다.

```
1 class Derived : public Base {
2     int data{10};
3     public:
4     Derived() : Base(), data(10)
5     virtual void vfunc() override{ cout << "Derived vfunc()" << data << endl; }
6 };
```

Base()가 먼저 호출되기 때문에 data의 정보를 사용하는 Derived의 vfunc를 호출하면 잘못된 정보를 호출하게 된다. 따라서 Base 생성자에서는 Base vfunc가 호출된다.

2.4 This call

```

1   class Point{
2     int x{0};
3     int y{0};
4     public:
5     void set(int a, int b) {
6       x=a; y=b;
7     }
8   };
9
10  int main(){
11    Point pt1;
12    Point pt2;
13
14    pt1.set(10,20);
15    pt2.set(10,20);
16 }
```

pt1, pt2를 생성하면 메모리 공간에 **멤버 변수가 객체 당 하나씩 생성된다.** 그렇다면 멤버 함수 set 또한 객체 당 하나씩 생성될까? 그렇지 않다. 멤버 함수는 객체가 여러개 생성되어도 **메모리에 한 개만** 만들어져 있다.

그렇다면 set 함수 인자가 a, b 밖에 없는데 x가 어떤 객체의 멤버인지 어떻게 알 수 있을까? 컴파일러가 이런 문제를 해결하기 위해 다음과 같이 컴파일 해준다.

```

1   class Point{
2     int x{0};
3     int y{0};
4     public:
5     void set(Point* this, int a, int b) {
6       this->x=a; this->y=b;
7     }
8   };
9   int main(){
10    Point pt1;
11    Point pt2;
12
13    set(pt1, 10,20);
14    set(pt2, 10,20);
15 }
```

사용자가 2개의 인자로 set를 구성했다하더라도 컴파일러가 객체 포인터 주소 인자가 같이 전달되도록 3개의 인자로 구성된 set 함수가 완성된다. 이러한 기술을 **this call**이라고 한다.

주의할 점은 실제 함수 인자가 전달되는 방식과 객체 주소가 전달되는 방식은 어셈블리 레벨에서는 약간 차이가 있다. 컴파일러마다 구현하는 방식도 다르니 위 예제 코드는 저런 개념으로 전달된다 정도만 숙지하면 된다.

static 멤버 함수는 this call이 적용되지 않는다.

```

1   class Point{
2     int x{0};
3     int y{0};
4     public:
5     void set(int a, int b) {
6       x=a; y=b;
7     }
8
9     static void foo(int a) {
10      x=a;
11    }
12  };
13
14  int main(){
15    Point pt1;
16    Point pt2;
17
18    pt1.set(10,20);
19    pt2.set(10,20);
20
21    Point::foo(10); // static 멤버함수는 객체 주소를 전달하지 않는다.
```

```
22     pt1.foo(10); // 실제로는 객체 주소 없이 Point::foo(10)으로 변환되어 실행된다.
23 }
```

2.5 Member function pointer

멤버 함수를 함수 포인터로 가르킬 수 있을까? 다음 예제를 보자.

```
1 class X {
2     public:
3         void mf1(int a) {}
4         static void mf2(int a) {}
5     };
6
7     void foo(int a) {}
8
9     int main() {
10         void(*f1)(int) = &foo;           // ok
11         void(*f2)(int) = &X::mf1; // error. this call에 의해 파라미터 개수 맞지 않음!
12         void(*f3)(int) = &X::mf2; // ok. static 멤버 함수는 this call이 없으므로 적용 가능
13 }
```

일반 함수 포인터에 **멤버 함수의 주소를 담을 수 없다**. 하지만 **static** 멤버 함수의 주소는 담을 수 있다. 그렇다면 멤버 함수의 주소를 담으려면 어떻게 해야 할까?

```
1 class X {
2     public:
3         void mf1(int a) {}
4         static void mf2(int a) {}
5     };
6
7     void foo(int a) {}
8
9     int main() {
10         void(*f1)(int) = &foo;
11         //void(*f2)(int) = &X::mf1;
12         void(*f3)(int) = &X::mf2;
13
14         void(X::*f2)(int) = &X::mf1; // ok. 멤버 함수 주소는 이렇게 담아야 한다.
15 }
```

일반 함수 포인터는 `void(*f1)(int)= foo` 또는 `&foo`를 입력해도 함수 주소로 암시적 변환이 가능하지만 멤버 함수 포인터는 `void(X::*f2)= X::mf1`으로 하면 컴파일 에러가 발생하므로 반드시 `&X::mf1`으로 주소를 넣어줘야 함에 유의한다.

멤버 함수 포인터는 일반 함수 포인터처럼 호출할 수 있을까?

```
1 f1(10); // ok
2 f2(10); // error. 객체가 필요하다.
```

멤버 함수 포인터는 객체가 있어야 사용 가능하다.

```
1 X obj;
2 obj.f2(10); // error. f2라는 멤버를 찾게 된다.
3
4 // pointer to member 연산자 사용
5 obj.*f2(10); // error. 연산자 우선순위 문제로 괄호가 먼저 계산된다.
6 (obj.*f2)(10); // ok
```

`.*`는 하나의 연산자 역할을 하며 **pointer to member operator**라고 부른다. 객체가 포인터인 경우 `->*`을 사용하면 된다.

```
1 (obj.*f2)(10); // ok
2 (pobj->*f2)(10); // ok
```

2.5.1 std::invoke, std::mem_fn

멤버 함수 포인터를 사용하는 형태가 너무 복잡해보인다. 이를 일반 함수 포인터처럼 조금 더 쉽게 호출할 수는 없을까?

```
1  class X {
2      public:
3          void mf1(int a) {}
4          static void mf2(int a) {}
5      };
6
7  void foo(int a) {}
8
9  int main() {
10     void(*f1)(int) = &foo;
11     void(X::*f2)(int) = &X::mf1;
12
13     X obj;
14
15     f1(10);           // 일반 함수 포인터 사용
16     (obj.*f2)(10); // 멤버 함수 포인터 사용
17     f2(&obj, 10); // 위 모양이 너무 복잡해서 이렇게 사용할 수는 없을까? 실제 논의가 있었다.
18     // 이를 uniform call syntax라고 한다.
19 }
```

하지만 **uniform call syntax**는 컴파일러가 어셈블리 레벨을 많이 바꿔야 하기 때문에 채택되지 않았다. 대신 STL에서는 몇 가지 도구를 제공한다.

std::invoke는 c++17부터 나왔으며 일반 함수 포인터와 멤버 함수 포인터(정확히는 **callable object**)를 정확히 동일한 방법으로 호출할 수 있다.

```
1 #include <functional>
2
3 std::invoke(f1, 10);
4 std::invoke(f2, obj, 10);
5 std::invoke(f2, &obj, 10);
```

또는 **std::mem_fn**을 사용해도 된다. 이는 c++11부터 나왔으며 멤버 함수 주소를 인자로 받아서 함수 주소를 담은 래퍼 객체를 반환한다.

```
1 #include <functional>
2
3 auto f3 = std::mem_fn(&X::mf1); // 반드시 auto로 받아야 한다.
4 f3(obj, 10);
5 f3(&obj, 10);
```

2.6 Member data pointer

일반 변수의 포인터처럼 멤버 변수의 포인터도 만들 수 있을까?

```
1 struct Point {
2     int x;
3     int y;
4 };
5
6 int main() {
7     int num=0;
8     int *p1 = &num; // ok
9
10    int Point::*p2 = &Point::y; // ok. 멤버 변수의 포인터는 이렇게 가르켜야 한다.
11 }
```

p1은 num 변수의 메모리 주소를 가지고 있다. 그런데 Point 타입의 객체가 존재하지 않는데 p2는 무엇을 담고 있을까? 대부분의 컴파일러는 Point 구조체 안에서 y의 offset을 담고 있다. 이는 공식 표준은 아니지만 대부분의 컴파일러가 이렇게 구현되어 있다.

멤버 변수 포인터는 다음과 같이 사용한다.

```
1 *p1 = 10; // ok
2 *p2 = 10; // error
3
4 Point pt;
5 pt.*p2 = 10; // ok. pt.y = 10
6 // *(&pt + p2) = 10; p2만큼 오프셋 주소를 더하여 그 곳의 변수에 10을 넣는다.
```

멤버 변수 포인터도 `invoke`를 통해 호출할 수 있다.

```
1 std::invoke(p, obj) = 10; // ok. obj.y = 10. 일반 함수 포인터처럼 사용할 수 있다.
2 int n = std::invoke(p, obj); // ok. 데이터를 꺼내는 것도 가능하다
```

`std::invoke`는 일반 포인터, 함수, 멤버 함수 포인터, 멤버 변수 포인터, 람다 표현식을 전부 일관되게 동작할 수 있게 해준다. 이런 타입들을 **callable type**라고 한다.

2.7 Implement custom max

이번 섹션에서는 임의의 `max` 함수를 작성해보자.

```
1 #include<iostream>
2 #include<string>
3
4 template<class T>
5 const T& mymax(const T& obj1, const T& obj2) {
6     return obj1 < obj2 ? obj2 : obj1;
7 }
8
9 int main() {
10     std::string s1 = "abcd";
11     std::string s2 = "xyz";
12
13     auto ret1 = mymax(s1, s2);
14     std::cout << ret1 << std::endl;
15 }
```

`mymax` 함수는 문자열의 처음 글자인 `a`와 `x`를 비교하여 `x`가 더 뒤에 있으므로 `s2`를 반환하게 된다. **만약 문자열의 순서가 아니라 문자열의 개수를 `mymax`를 통해 비교하고 싶다면 어떻게 해야 할까?**

이와 같이 알고리즘 함수가 사용하는 "정책(비교 방식)"을 변경하고 싶은 경우 다음과 같은 방법을 사용 할 수 있다.

- 이항 조건자(binary predicator) 사용 (e.g., c++ stl)

```
1     std::sort(v.begin(), v.end(), [](auto &a, auto &b) { return a.size() < b.size(); });
```

- 단항 조건자(unary predicator) 사용 (e.g., python)

```
1     sorted(str_list, key = lambda x : len(x));
```

- 단항, 다항 조건자를 모두 사용 (e.g., c++ 20 range 알고리즘의 원리). 멤버 함수 포인터, 멤버 데이터 포인터, `std::invoke`를 모두 활용한다.

```
1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return proj(obj1) < proj(obj2) ? obj2 : obj1; // proj()를 통해 비교하고 결과값은 원래 값을 반환한다.
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); });
11    std::cout << ret1 << std::endl;
```

`mymax`의 3번째 인자로 단항 조건자를 전달하면 비교 시 조건자의 결과를 비교한다. 이는 c++20에서 **Projection**이라는 기술로 불린다.

만약 단항 조건자 대신 멤버 함수 포인터를 전달할 수는 없을까?

```

1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return proj(obj1) < proj(obj2) ? obj2 : obj1;
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); });
11    auto ret2 = mymax(s1, s2, &std::string::size); // error. 멤버 함수 포인터이므로 proj(obj)에서
12        // 에러가 발생한다.
13 }
```

`proj(obj)`는 일반 함수라면 `ok`이지만 멤버 함수라면 에러가 발생한다. 이 때, `std::invoke`를 사용하면 일반 함수와 멤버 함수 모두 커버할 수 있다.

```

1 template<class T, class Proj>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1; // 일반 함수, 멤버 함수
4         // 모두 커버 가능
5 }
6
7 int main() {
8     std::string s1 = "abcd";
9     std::string s2 = "xyz";
10
11    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); }); // ok
12    auto ret2 = mymax(s1, s2, &std::string::size); // ok.
13 }
```

Projection은 생략될 수 있어야 한다. 이를 위해서는 `Proj`의 디폴트 값이 있어야 한다. `std::identity`는 전달 받은 인자를 어떤 변화 없이 참조값을 반환하는 함수 객체이다. 이를 활용하여 디폴트 값을 정의한다.

```

1 template<class T, class Proj = std::identity> // std::identity로 디폴트 값 설정
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4 }
5
6 int main() {
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2, [](auto &a) { return a.size(); }); // ok
11    auto ret2 = mymax(s1, s2, &std::string::size); // ok.
12    auto ret3 = mymax(s1, s2); // ok
13 }
```

`std::identity`은 c++20에 처음 등장하였으므로 구현 코드는 다음과 같이 되어 있다. `<functional>` 헤더 파일이 필요하다.

```

1 struct identity {
2     template<class _Ty>
3     [[nodiscard]] constexpr
4     _Ty&& operator() (_Ty&& arg) const noexcept {
5         return std::forward<_Ty>(arg);
6     }
7     using is_transparent = int;
8 }
```

다음으로 mymax를 통해 임의의 객체 Point를 비교해보자.

```
1 template<class T, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4     // (obj1.*proj) < (obj2.*proj)
5 }
6
7 struct Point {
8     int x,y;
9 }
10
11 int main() {
12     Point p1 = {2,0};
13     Point p2 = {1,1};
14
15     auto ret = mymax(p1, p2, &Point::y); // 이렇게 비교할 수는 없을까?
16     cout << ret.x << ", " << ret.y << endl;
17 }
```

`std::invoke`는 멤버 변수의 포인터도 커버할 수 있다. 따라서 위 코드는 아무 수정 없이 정상적으로 동작한다. 정리하면 `mymax`은 다음과 같이 4가지 사용법이 존재한다.

```
1 string s1 = "abcd";
2 string s2 = "xyz";
3
4 auto ret1 = mymax(s1, s2); // (1)
5 auto ret2 = mymax(s1, s2, [](auto &a) { return a.size(); }); // (2)
6 auto ret3 = mymax(s1, s2, &std::string::size); // (3)
7
8 Point p1 = {0,0};
9 Point p2 = {1,1};
10
11 auto ret4 = mymax(p1, p2, &Point::y); // (4)
```

2.7.1 Add compare operator

앞서 살펴본 `mymax` 함수는 `std::invoke`를 사용하여 여러 케이스에 대해 사용 가능한 훌륭한 함수였다.

```
1 template<class T, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Proj proj = {}) {
3     return std::invoke(proj, obj1) < std::invoke(proj, obj2) ? obj2 : obj1;
4 }
```

`mymax` 함수는 `<` 연산을 기본으로 하고 있다. 이를 유저가 원하는 임의의 비교 연산자를 사용할 수는 없을까? 세번째 인자에 `Comp` 연산자를 추가함으로서 이를 가능하게 할 수 있다.

```
1 template<class T, class Comp = std::less<void>, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Comp comp = {}, Proj proj = {}) {
3     return std::invoke(comp, std::invoke(proj, obj1), std::invoke(proj, obj2)) ? obj2 : obj1;
4     // comp가 멤버 함수일 때는 comp(a,b)가 불가능하기 때문에 std::invoke(comp, a, b)로 작성해준다.
5 }
6
7 int main(){
8     std::string s1 = "abcd";
9     std::string s2 = "xyz";
10
11     auto ret1 = mymax(s1, s2);
12     auto ret2 = mymax(s1, s2, std::greater{});
13     auto ret2 = mymax(s1, s2, {}, &std::string::size); // 세번째 {}는 디폴트 연산자를 사용하라는 의미
14     auto ret2 = mymax(s1, s2, std::greater{}, &std::string::size);
15 }
```

왜 템플릿의 디폴트 인자를 `std::less<T>`가 아닌 `std::less<void>`로 했을까? 이는 callable section을 공부하고 오면 답을 알 수 있다.

2.7.2 c++20 ranges algorithm

지금까지 구현한 `mymax`는 c++20에서 도입된 range algorithm 기반의 `std::ranges::max` 함수와 거의 동일한 구현 코드를 갖는다. 하지만 이번 섹션에서 만든 `mymax`은 함수 템플릿이지만 `std::ranges::max`은 함수 객체(템플릿)임에 유의한다.

```
1 template<class T, class Comp = std::less<void>, class Proj = std::identity>
2 const T& mymax(const T& obj1, const T& obj2, Comp comp = {}, Proj proj = {}) {
3     return std::invoke(comp, std::invoke(proj, obj1), std::invoke(proj, obj2)) ? obj2 : obj1;
4 }
5
6 int main(){
7     std::string s1 = "abcd";
8     std::string s2 = "xyz";
9
10    auto ret1 = mymax(s1, s2);
11    auto ret2 = mymax(s1, s2, std::greater{}, &std::string::size);
12    auto ret3 = std::ranges::max(s1, s2, std::ranges::greater, &std::string::size); // ret2와 동일한
13                                결과 출력
}
```

c++20의 range algorithm은 알고리즘의 "비교 정책"을 교체할 수 있으며 "Projection"을 전달할 수 있다. 이 때 `std::invoke`를 통해 구현하여 멤버 함수 포인터와 멤버 데이터 포인터 모두 사용이 가능하다. 또한 반복자 구간이 아닌 컨테이너를 전달받기 때문에 다음과 같이 조금 더 편하게 코딩이 가능하다.

```
1 std::vector<std::string> v = {"hello", "a", "xxx", "zz"};
2
3 std::sort(v.begin(), v.end()); // 매 번 begin, end를 적어줘야 하므로 불편함
4
5 // c++20 ranges
6 std::ranges::sort(v); // 컨테이너만 넘겨주면 자동으로 정렬. 디폴트로 알파벳 순서대로 정렬
7 std::ranges::sort(v, std::ranges::greater{}, &std::string::size); // 비교 정책과 Projection 모두
8 전달 가능
```

2.8 Size of member function pointer

이번 섹션에서는 멤버 함수 포인터의 크기를 알아본다. 이를 알기 위해서는 우선 다중상속을 정확히 이해하고 있어야 한다.

```
1 struct A {int x;};
2 struct B {int y;};
3 struct C : public A, public B { // 다중 상속
4     int z;
5 };
6
7 int main(){
8     C cc;
9     cout << &cc << endl; // 1000
10
11    A* pA = &cc;
12    B* pB = &cc;
13    cout << pA << endl; // 1000
14    cout << pB << endl; // 1004. B클래스는 두번째로 상속을 받고 있으므로 A 클래스의 멤버 변수가 차지하는 4
15                                바이트를 견너뛰어 1004의 값을 갖는다
}
```

`static_cast`를 해도 똑같이 1004의 값이 나온다. `reinterpret_cast`를 사용해야 1000의 값이 나온다. `reinterpret_cast`은 0x1000 위치의 메모리를 B타입처럼 사용하겠다라는 의미이다. 메모리를 다르게(다른 타입으로) 사용하겠다는 의미이므로 사용에 주의해야 한다.

```
1 B* pB1 = static_cast<B*>(&cc); // 1004
2 B* pB2 = reinterpret_cast<B*>(&cc) // 1000
```

다음으로 멤버 함수가 있는 예제를 보자.

```

1  struct A{
2      int x;
3      void fa() { std::cout << this << std::endl; }
4  };
5  struct B{
6      int y;
7      void fb() { std::cout << this << std::endl; }
8  };
9  struct C : struct A, struct B{
10     int z;
11     void fc() { std::cout << this << std::endl; }
12 };
13
14 int main() {
15     C cc;
16     cc.fc(); // 0x1000
17     cc.fa(); // 0x1000
18     cc.fb(); // 0x1004. 이전 코드와 동일하게 B에 대한 상속이 두번째로 받았으므로 메모리 공간에 4바이트만큼
19     뒤로 간 1004가 출력된다.
}

```

다음과 같이 함수 포인터가 있다고 하자.

```

1 void (C::*f)();
2
3 f = &C::fa;
4 (cc::*f)(); // 1000
5
6 f = &C::fb;
7 (cc::*f)(); // 1004. 정상적으로 함수 포인터가 동작한다.

```

함수 포인터는 런타임 시점에서 `f(&cc)`와 같이 호출된다. 하지만 `f = &C::fb`를 가리키는 경우 1004가 정상적으로 호출되려면 `f(&cc + sizeof(A))`가 호출되어야 한다. **컴파일 타임에선 f가 fa를 가리킬지 fb를 가리킬지는 알 수 없는데 어떻게 정상적으로 동작하는 것일까?**

멤버 함수 포인터는 항상 4byte(32bit), 8byte(64bit)인 것은 아니다. 코드에 따라 멤버 함수 포인터의 크기가 달라지는데 **다중 상속의 경우 함수 주소와 this offset을 같이 보관하여 8byte(32bit), 16byte(64bit)의 크기를 가진다.**

```

1 void(C::*f)(); // 함수 주소 + this offset을 함께 보관 8byte(32bit), 16bit(64bit)
2
3 f = &C::fa; // fa주소, 0이 보관
4 f = &C::fb; // fb주소, sizeof(A) 보관

```

위와 같은 동작이 c++ 표준 동작은 아니다. 컴파일러마다 원리가 다를 수 있으나 대부분의 컴파일러가 위와 같은 방법을 통해 멤버 함수 포인터를 관리한다.

Tip

`void*`가 모든 주소를 담는다고 알려져 있지만 엄밀하게 보면 {주소, this offset}을 보관하는 **멤버 함수의 포인터는 담을 수 없다.** 또한 멤버 데이터의 포인터도 진짜 메모리 주소가 아닌 상대적인 offset 정보만 가지고 있기 때문에 **멤버 데이터 포인터 또한 담을 수 없다.**

```

1 struct myostream{
2     myostream& operator<<(int n) { printf("int : %d\n", n); return *this; }
3     myostream& operator<<(double d) { printf("double : %d\n", n); return *this; }
4     myostream& operator<<(bool b) { printf("bool %d\n", n); return *this; }
5     myostream& operator<<(void* p) { printf("void*: %d\n", n); return *this; }
6 }
7 myostream mycout;
8
9 int main() {
10     int n=10;
11     double d=3.4;

```

```

12     int Point::*p = &Point::y;
13
14     mycout << n;      // int
15     mycout << d;      // double
16     mycout << &n; // void*
17     mycout << &d; // void*
18     mycout << p; // 1. 1이 출력되는 이유는 void*로 변환될 수 없는 멤버 변수 포인터가 bool로는 암시적
19         변환이 되기 때문에 4 -> 1(true)가 되어 1이 출력되게 되는 것이다.
}

```

따라서 멤버 함수의 포인터를 출력해보고 싶을 때는 cout보단 printf를 써서 출력하는 것이 좋다.

2.9 new, delete and placement new

이번 섹션에서는 `new`, `delete` 키워드에 대해서 자세히 살펴본다.

```

1 class Point{
2     int x,y;
3     public:
4     Point(int a, int b) : x{a}, y{b} { cout << "Point(int,int)" << endl; }
5     ~Point() { cout << "~Point()" << endl; }
6 }
7
8 int main(){
9     Point* p1 = new Point(1,2); // 메모리 할당, 생성자 호출
10    delete p1; // 소멸자 호출, 메모리 해제
11 }

```

c++에서 `new` 키워드를 사용하면 다음과 같은 메모리 할당, 생성자 호출 코드가 자동으로 수행된다.

```

1 Point* p1 = new Point(1,2);
2
3 void* p = operator new(sizeof(Point)); // 1. 메모리 할당
4 Point* p1 = new(p) Point(1,2); // 2. 생성자 호출. new(p)를 placement new라고 부른다.

```

`delete`를 사용하면 소멸자가 호출되고 메모리가 해제되는 코드가 자동으로 수행된다.

```

1 delete p1;
2
3 p1->~Point(); // 1. 소멸자 호출
4 operator delete(p1); // 2. 메모리 해제

```

만약 생성자 호출 없이 메모리만 할당, 해제하고 싶은 경우 다음과 같이 쓰면 된다. c언어에서 `malloc`과 `free`한 것과 동일하다.

```

1 void* p = operator new(sizeof(Point)); // Point 구조체는 x,y로 인해 8바이트 할당
2
3 operator delete(p); // 메모리 해제

```

`operator new`, `delete` 함수는 c++20 기준으로 각각 22개, 30개의 다른 버전이 있다. 두 함수는 `<new>` 헤더를 필요로 하며 std namespace가 아닌 global namespace에 존재한다.

```

1 [[nodiscard]] void* operator new(std::size_t);
2 void operator delete (void* ptr) noexcept;

```

앞서 메모리를 할당한 다음 생성자를 호출하려면 아래와 같이 하면 된다.

```

1 void *p1 = operator new(sizeof(Point));
2 Point *p2 = new(p1) Point(1,2); // 생성자 호출

```

`new(p1)`은 **placement new**라고 불리며 메모리 할당없이 이미 할당된 메모리에 생성자를 명시적으로 호출하기 위한 `new`이다. c++20에서는 `new(p) Point(1,2)` 대신 `std::construct_at(p, 1, 2)`를 통해 생성자를 명시적으로 호출할 수 있다. 유사하게 `std::destroy_at(p)`를 통해 소멸자를 호출할 수 있다.

`void*`가 아닌 정확한 클래스 타입으로 메모리를 할당받기 위해서는 다음과 같이 작성한다.

Tip

- `new Point(1,2);` // 새로운 메모리를 할당하고 객체 생성
- `new(p) Point(1,2);` // 이미 할당된 메모리(p)에 객체 생성 (=placement new)
- `std::construct_at(p, 1, 2);` // 위와 동일. c++20, <memory>
- `std::destroy_at(p);` // 명시적 소멸자 호출

```
1 Point* p3 = static_cast<Point*>(operator new(sizeof(Point)));
2 std::construct_at(p3, 1, 2); // 생성자 호출
```

2.9.1 Using placement new

메모리 할당과 생성자 호출을 분리해야 하는 이유가 있을까?

```
1 class Point{
2     int x,y;
3     public:
4     Point(int a, int b) :x{a}, y{b} {} // 디폴트 생성자가 없음에 주목!
5     ~Point() {}
6 };
7
8 int main(){
9     Point* p1 = new Point(0,0); // Point 객체 한개를 힙에 생성하고 싶은 경우
10
11    Point* p2 =? // 만약 Point 객체 3개를 연속적인 형태(배열 형태)로 생성하고 싶은 경우 어떻게 해야 할까?
12 }
```

p2를 `new Point[3]`와 같이 생성하면 Point 타입에는 반드시 디폴트 생성자가 있어야 한다. 하지만 **디폴트 생성자 없이도 3개를 연속적인 배열 형태로 만들고 싶은 경우도 발생할 수 있다.**

c++11에는 다음과 같이 초기화할 수 있다. `{0,0}`은 인자 2개짜리 생성자를 호출하기 때문에 예러없이 잘 빌드된다.

```
1 Point* p2 = new Point[3]{{0,0}, {0,0}, {0,0}};
```

하지만 3개가 아니라 30개, 100개가 넘어가는 경우는 어떻게 해야 할까? 이럴 때는 **메모리 할당과 생성자 호출을 분리하면 훨씬 편하게 생성할 수 있다.**

```
1 Point* p2 = static_cast<Point*>(operator new(sizeof(Point)*3));
2
3 for(int i=0; i<3; i++){
4     new(&p2[i]) Point(0,0); // c++20 이전 표기법
5     std::construct_at(&p2[i], 0,0); // c++20 이후 표기법
6 }
7
8 for(int i=0; i<3; i++){
9     p2[i].~Point(); // c++20 이전 표기법
10    std::destroy_at(&p2[i]); // c++20 이후 표기법
11 }
```

2.9.2 vector and placement new

```
1 int main(){
2     vector<int> v(10);
3
4     v.resize(7); // size의 크기를 줄이고 메모리 할당 크기는 capacity 변수에 저장한다.
5     cout << v.size() << endl; // 7
6     cout << v.capacity() << endl; // 10
```

```

7     v.resize(8);
8     cout << v.size() << endl; // 8
9     cout << v.capacity() << endl; // 10
10 }

```

벡터가 임의의 클래스 X를 담고 있다고 해보자.

```

1 struct X{
2     X() { cout << "X() get resource" << endl; }
3     ~X() { cout << "~X() release resource" << endl; }
4 }
5
6 int main(){
7     vector<X> v(10);
8
9     v.resize(7); // 크기가 줄어드는 경우 메모리는 제거하지 않더라도 소멸자는 불러야하지 않을까?
10
11    v.resize(8); // 8번째 메모리에 생성자가 호출되어야 하지 않을까?
12 }

```

사용자 정의 타입을 vector에 담으면 **메모리의 할당, 해지가 없는데도 불구하고 생성자, 소멸자의 호출을 정말 많이 호출**한다. 따라서 메모리 할당, 생성자 호출에 대한 개념의 정확한 이해가 필요하다.

2.10 Trivial constructor

2.10.1 Trivial default constructor

생성자가 trivial하다는 말은 컴파일러가 자동으로 생성해주면서 동시에 아무 일도 하지 않을 때를 말한다

2.10.2 Trivial copy constructor

복사 생성자가 trivial하다는 말은 멤버변수 값을 복사하는 것 이외에 아무 일도 하지 않을 때를 말한다. 복사 생성자가 trivial하다면 배열 전체를 `memcpy`, `memmove` 등으로 복사하는 것이 빠르다! 복사 생성자가 trivial하지 않다면 배열의 모든 요소에 대해 하나씩 “복사 생성자”를 호출해서 생성자를 호출해야 한다

```

1 struct Point {
2     int x=0;
3     int y=0;
4 };
5
6 template<class T>
7 void constexpr copy_type(T* dst, T* src, std::size_t sz) {
8     if(std::is_trivially_copy_constructible_v<T>) {
9         std::cout << "using memcpy" << std::endl;
10        memcpy(dst, src, sizeof(T)*sz);
11    }
12    else {
13        std::cout << "using copy ctor" << std::endl;
14        while(sz--){
15            new(dst) T(*src);
16            --dst, --src;
17        }
18    }
19 }
20
21 int main(){
22     Point arr1[5];
23     Point arr2[5];
24     copy_type(arr1, arr2, 5);
25 }

```

위 코드에서 Point 클래스는 `int x,y`와 같이 간단한 멤버변수만 존재하므로 trivial copy constructor이다. 하지만 `virtual void foo(){}` 같이 가상함수를 사용하거나 `string s;`와 같이 복사하는 클래스를 사용하게 되면 trivial하지 않게 된다. 이런 경우에는 `placement new` 또는 `std::construct_at`을 사용해야 한다.

2.11 Type deduction

컴파일 타입에 타입이 결정되는 `auto` 키워드에 대해 살펴보자

```
1 int main(){
2     int n=10;
3     const int c =10;
4
5     auto a1 = n; // int a1=n;
6     auto a2 = c; // (1) const int a2 = c; --> no.
7     // (2) int a2 = c; --> ok.
8 }
```

type deduction(타입 추론)이 발생하는 키워드는 다음과 같다: `template`, `auto`, `decltype`

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << typeid(T).name() << std::endl;
4 }
5 int main(){
6     int n=10;
7     foo(n); // T=int
8     foo<const int&>(n); // T=const int&. But typeid(T).name() keep printing output 'int'
9 }
```

`typeid(T).name()`은 타입 이름만 추론할 뿐 `const`, `volatile`, `reference` 정보가 출력되지 않는다.

1. 이럴 때는 의도적으로 에러를 발생시켜서 정확한 타입을 에러 메시지를 통해 알 수 있다.
2. 또는 `boost::type_index` 라이브러리에 `type_id_with_cvr<T>().pretty_name()`을 사용하면 됨
3. 컴파일러가 제공하는 매크로를 사용하면 된다.
 - `__FUNCTION__`: 함수의 이름만 보여주므로 타입 추론에는 사용하지 않음
 - `__PRETTY_FUNCTION__`: g++, clang에서는 함수 이름과 타입이 나옴
 - `__FUNCSIG__`: cl.exe 버전

```
1 std::cout << __FUNCTION__ << std::endl;
2 std::cout << __PRETTY_FUNCTION__ << std::endl; // g++, clang
3 std::cout << __FUNCSIG__ << std::endl; // cl.exe
```

T가 값인 경우를 살펴보자

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     while(--arg>0) {}
4 }
5
6 int main(){
7     int n=10;
8     int& r = n;
9     const int c = 10;
10    const int& cr = c;
11    foo(n); // T=int
12    foo(r); // T=int& 일 것 같지만 T=int
13    foo(c); // T=const int 일 것 같지만 T=int
14    foo(cr); // T=const int& 일 것 같지만 T=int
15 }
```

T 인자를 값으로 받을 때는 복사본 객체가 만들어져서 “`const`, `volatile`, `reference`” 속성을 제거하고 값만 받는다. 헷갈리는 것 중 하나가 값으로 받을 때는 인자의 `const` 속성은 제거되고 “인자가 가리키는 곳의 `const` 속성은 유지”한다. 무슨 이야기인지 살펴보자

```
1 #include <iostream>
2 template<class T> void foo(T arg){
3     std::cout << __PRETTY_FUNCTION__ << std::endl;
```

```

4   }
5   int main(){
6     const char* const s = "hello";
7     foo(s); // 포인터의 const 속성은 제거되지만 가리키는 곳 "hello"의 const 속성은 유지됨!
8     // const char* arg = "hello"가 됨!!
9   }

```

T 인자를 참조로 받을 때는 다음과 같다.

```

1 #include <iostream>
2 template<class T> void foo(T& arg){
3   std::cout << __PRETTY_FUNCTION__ << std::endl;
4 }
5
6 int main(){
7   int n = 10;           // T=int.      arg=int&
8   int& r = n;          // T=const int. arg=const int& (const 속성은 유지!)
9   const int c = 10;    // T=int        arg=int&. (T에서 reference 속성은 제거!)
10  const int& cr = c; // T=const int  arg=const int& (const 속성은 유지!)
11 }

```

주의해야 할 점은 T의 타입과 arg의 타입은 다르다는 것이다 (T& arg이기 때문!). 함수 인자의 “reference를 제거하고 T의 타입을 결정한다”. 인자가 가진 “`const, volatile` 속성은 유지한다.”. 마지막으로 T에 배열이 전달된 경우를 살펴보자

```

1 template<class T> void foo(T arg){
2   std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5   std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9   int x[3] = {1,2,3};
10  foo(x); // T=int* 타입으로 받는다
11  goo(x); // T=int[3] 타입으로 받는다. arg는 int() [3] 타입으로 받는다
12 }

```

T& arg로 배열을 받는 경우 `int (&arg)[3] = x;`; 처럼 받는게 되어서 배열의 reference가 된다. 따라서 아래와 같이 goo()를 사용하면 에러가 발생한다.

```

1 template<class T> void foo(T arg){
2   std::cout << __PRETTY_FUNCTION__ << std::endl;
3 }
4 template<class T> void goo(T& arg){
5   std::cout << __PRETTY_FUNCTION__ << std::endl;
6 }
7
8 int main(){
9   foo("orange", "apple"); // ok
10  goo("orange", "apple"); // error
11 }

```

`foo`와 `goo` 둘 다 `const char` 형식으로 받지만 포인터는 개수에 제한이 없으므로 ok인 반면에 reference는 `const char[7]`, `const char[6]`은 다른 reference이기 때문에 같은 T를 받는 상황에서 에러가 발생한다!

`foo(const char[7], const char[6])`, `goo(const char[7], const char[6])`

2.11.1 Auto type deduction

`template`은 함수 인자로 추론하는 반면 `auto`는 우변의 표현식으로 타입을 추론한다. `template`을 ‘T arg = parameter’처럼 추론한다고 볼 수 있으므로 사실 ‘`auto a = expression`’과 동일한 형태로 추론한다!

```

1 int main(){
2   int n=10;
3   int& r = n;

```

```

4     const int c = 10;
5     const int& cr = c;
6
7     auto a1 = n; // auto=int
8     auto a2 = r; // auto=int
9     auto a3 = c; // auto=int
10    auto a4 = cr; // auto=int
11
12    auto& a5 = n; // auto=int. a5=int&
13    auto& a6 = r; // auto=int. a6=int&
14    auto& a7 = c; // auto=const int. a7=const int&
15    auto& a8 = cr; // auto=const int. a8=const int&
16
17    int x[3] = {1,2,3};
18    auto a = x; // auto=int*
19    auto& b = x; // auto=int[3]. b=int(&) [3]
20
}
```

위와 같이 template에서 본 규칙들이 그대로 적용! 아래와 같이 조금 더 까다로운 경우를 보자.

```

1 int main(){
2     auto a1 = 1;
3     auto a2 = {1};
4     auto a3{1};
5
6     std::cout << typeid(a1).name() << std::endl; // auto=int
7     std::cout << typeid(a2).name() << std::endl; // auto=initialized_list
8     std::cout << typeid(a3).name() << std::endl; // auto=int
9     std::vector<int> v1(10,0);
10    std::vector<bool> v2(10,false);
11
12    auto a4 = v1[0];
13    auto a5 = v2[0];
14
15    std::cout << typeid(a4).name() << std::endl; // auto=int
16    std::cout << typeid(a5).name() << std::endl; // auto=temporary proxy 객체
17 }
```

배열은 `auto a= {1}`라고 하면 배열 타입으로 추론된다. `bool` 타입은 최적화 과정에서 specialization되어 있다. 따라서 `bool`은 [] 연산자가 bool로 변환 가능한 `temporary proxy`으로 추론한다!

2.11.2 Array Name

배열의 이름은 배열의 1번째 요소의 주소로 암시적 형변환 된다.

```

1 int main(){
2     int x[3] = {1,2,3};
3
4     int *p0[3] = &x; // error. 연산자 우선 순위에 따라 p[3], *p 순으로 추론된다.
5     int (*p1)[3] = &x; // ok. (*p)로 감싸줘야 x[3] 배열에 대한 제대로된 포인터가 된다.
6     int *p2 = x; // ok. &x[0]
7
8     printf("%p, %p\n", p1, p1+1); // 배열 자체를 가리키므로 +1을 하면 12바이트만큼 증가한다.
9     printf("%p, %p\n", p2, p2+1); // 배열의 첫번째 원소를 가리키므로 +1을 하면 4바이트만큼 증가한다.
10
11    (*p1)[0] = 10;
12    *p2 = 10;
13 }
```

위 코드에서 보면 `p1`, `p2`가 가리키는 주소는 동일하지만 포인터 타입이 다르므로 주소를 해석하는 방식이 다르다. 다음과 같이 배열을 함수 인자를 받는 경우에 대해 알아보자

```

1 void f1(int p[3]) {
2     printf("%d\n", sizeof(p)); // 마치 3개 배열을 입력으로 받는 듯 보이지만 int *p와 동일한 포인터를 받고
3     // 있다. 즉, 12가 아닌 8(64bit)이 나온다
4 }
```

```

4
5     int main(){
6         int x[3] = {1,2,3};
7         f1(x);
8     }

```

함수 인자로 배열을 받을 때는 `int *p`와 같이 포인터 타입으로 받거나 `int p[]`와 같이 배열 타입으로 받는다. 이 때 컴파일러에 의해 둘 다 **포인터**로 변환한다. (`int *p`).

`int p[3]`과 같이 배열의 크기도 지정해줄 수 있는데 이 또한 **포인터**로 컴파일러가 변환한다(`int *p`). 헷갈리기 쉬운 문법이니 주의한다. 마지막으로 다차원 배열의 포인터에 대해 알아보자

```

1     void foo(int (*p)[2]) {
2         p[0][0] = 100;
3     }
4     int main(){
5         int y[3][2] = {1,2,3,4,5,6};
6
7         int (*p3)[3][2] = &y; // 배열의 변수와 정확히 동일한 형태를 유지하고 (*p)로 감싸주면 배열 포인터가
8         된다.
9         int (*p4)[2] = y;    // 배열의 첫번째 원소는 2차원 배열이므로 (=1,2) int (*p4)[2]와 같이 선언해줘야
10        한다.
11
12     foo(y);
13 }

```

2.12 Rvalue & forwarding & reference

2.12.1 Lvalue vs Rvalue

- lvalue: 등호의 왼쪽에 올 수 있는 표현식
- rvalue: 등호의 왼쪽에 올 수 없는 표현식

이외에도 c++에서는 다음과 같은 추가적인 구분 방법이 존재한다.

```

1     x=10;
2     int f1() { return x; }
3     int& f2() { return x; }
4
5     int main(){
6         int v1=0, v2=0;
7
8         v1=10; // ok. v1 : lvalue
9         10=v1; // error. 10 : rvalue
10        v2=v1;
11        int *p1 = &v1; // ok
12        int *p2 = &10; // error
13
14        f1() = 20; // error
15        f2() = 20; // ok
16        const int c = 10; // 상수도 lvalue의 특징을 가지고 있다 (이름, 주소)
17        c = 20; // error
18        "aa"[0] = 'x'; // error. lvalue 문제가 아니라
19        // const char[3]이므로 에러 발생!
20    }

```

이름, 주소가 있으면 lvalue, 없으면 rvalue. 참조를 반환하면 lvalue, 값을 반환하면 rvalue라고 볼 수 있다. 다음과 같은 의문이 들 수 있다.

1. **모든 상수는 rvalue인가?** : 상수는 immutable lvalue로 취급한다.
2. **모든 rvalue는 상수인가?** : `Point(1,2).set(10,20)` 같이 temporary 객체도 멤버 변수를 호출할 수 있으므로 상수가 아니다

lvalue, rvalue에 대한 혼란 오해 중 하나가 객체, 변수에 부여되는 속성으로 오해한다. **하지만 이는 표현식 (expression)에 부여되는 속성이다!** 표현식이란 “하나의 값”을 만들어내는 코드 집합을 말한다!

```

1 int main(){
2     int n=3;
3
4     n=10;      // ok
5     n+2 = 10; // error. n+2=5인데 이는 값이므로 rvalue!
6     n+2*3 = 10; // error.
7
8     (n=20) = 10; // ok. 표현식 ok
9
10    ++n = 10; // ok
11    n++ = 10; // error. n=3-->4로 바뀌는데 이는 값이므로 error
12 }
```

어떤 값이 lvalue인지 rvalue인지 조사하려면 **decltype**을 사용하면 된다!

```

1 int main(){
2     int n = 10;
3
4     if(std::is_lvalue_reference_v<decltype(n++)>) // n++: rvalue, ++n: lvalue
5         std::cout << "lvalue" << std::endl;
6     else
7         std::cout << "rvalue" << std::endl;
8
9     if(std::is_lvalue_reference_v<decltype((n))>) // 그냥 n을 넣으면 rvalue로 잘못나온다. (n)을 통해
10    괄호로 감싸줘야 표현식으로 인식해서 lvalue로 정상 인식한다!
11    std::cout << "lvalue" << std::endl;
12 else
13     std::cout << "rvalue" << std::endl;
14 }
```

다음과 같이 매크로를 만들어 놓으면 편하게 구분할 수 있다.

```

1 #define value_category(...)
2 if( std::is_lvalue_reference_v<decltype((__VA_ARGS__))> )
3     std::cout << "lvalue" << std::endl;
4 else if( std::is_rvalue_reference_v<decltype((__VA_ARGS__))> )
5     std::cout << "rvalue(xvalue)" << std::endl;
6 else
7     std::cout << "rvalue(prvalue)" << std::endl;
8
9 int main(){
10     int n=10;
11
12     value_category(n);
13     value_category(n+2);
14     value_category(n++);
15     value_category(++n);
16 }
```

2.12.2 Reference & Overloading

참조자(reference)의 규칙에 대해 알아보자

```

1 int main(){
2     int n=3;
3
4     int& r1 = n; // ok
5     int& r2 = 3; // error
6     const int& r3 = n; // ok
7     const int& r4 = 3; // ok (하지만 상수성이 추가됨)
8
9     int&& r5 = n; // error
10    int&& r6 = 3; // ok. (상수 성질 없음! rvalue reference라고 부름)
11 }
```

임의의 숫자를 가리키고 싶을 땐 `const int& r4 = 3`과 같이 가리켜야 했으나 c++11 이후 rvalue-reference라는 문법이 등장하면서 `int&& r6=3`과 같이 가리킬 수 있게 되었음.

기존 `int& r1`을 lvalue-reference라고 부르며 `int&& r6=3`을 rvalue-reference라고 부름! 그렇다면 왜 상수성 없이 rvalue를 가리키는 것이 중요할까? 이는 move semantics와 perfect forwarding을 위해서 필요하다. 자세한 내용은 추후 다룰 예정이다. 다음으로 rvalue, lvalue의 함수 오버로딩에 대해 알아보자

```
1 class X{};  
2  
3 // void foo(X x) { std::cout << "X" << std::endl }. // 값 타입과 참조 타입은 서로 오버로딩될 수 없다!  
4 void foo(X& x) { std::cout << "X&" << std::endl } // 1  
5 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2  
6 void foo(X&& x) { std::cout << "X&&" << std::endl } // 3  
7 //void foo(const X&& x) { std::cout << "const X&&" << std::endl }  
8  
9 int main(){  
10     X x;  
11     foo(x); // lvalue. 어느 곳에 오버로딩 해야할지 몰라서 에러 발생  
12     // 값 타입을 주석처리하면 X x에 오버로딩 됨. (1, 2) 순서로 오버로딩  
13     foo(X()); // rvalue. 어느 곳에 오버로딩 해야할지 몰라서 에러 발생  
14     // 값 타입을 주석처리하면 X x에 오버로딩 됨. (3, 2) 순서로 오버로딩  
15 }
```

`foo` 함수의 마지막 표기법 `const X&& x`는 문법적으로는 가능하지만 사용하지 않는다. Move semantic을 통해 대체할 수 있기 때문이다. 다음은 주로 헷갈리는 문법에 대해 알아보자.

```
1 class X{};  
2  
3 void foo(X& x) { std::cout << "X&" << std::endl } // 1  
4 void foo(const X& x) { std::cout << "const X&" << std::endl }. // 2  
5 void foo(X&& x) { std::cout << "X&&" << std::endl } // 3  
6  
7 int main(){  
8     foo( X() ); // 3  
9  
10    X&& rx = X();  
11    foo(rx); // 3번이 호출될 것 같지만 1번이 호출됨! lvalue로 인식하기 때문  
12    foo(static_cast<X&&>(rx)); // 3  
13 }
```

`X&& rx = X()`에서 `rx`는 rvalue라고 생각할 수 있으나 이름이 있으므로 lvalue로 취급된다. 따라서 함수를 호출하면 1번이 호출된다!

따라서 3번을 호출하고 싶으면 `static_cast<X&&>()`을 사용하여 형변환을 하면 되는데 자세히 보면 같은 타입을 왜 변환해야 하는가? 하는 의문이 생길 수 있다. 이는 특수한 케이스로써 c++ 문법에서 타입 캐스팅이 아닌 value를 변환하는 캐스팅으로 기재되어 있다.

2.12.3 Reference collapsing

```
1 int main(){  
2     int n=3;  
3     int& lr = n; // lvalue reference  
4     int&& rr = 3; // rvalue reference  
5  
6     int& &ref2ref = lr; // error! 명시적으로 레퍼런스를 가리키는 레퍼런스는 코딩 불가  
7  
8     decltype(lr)& r1 = ? // int& & ==> int&  
9     decltype(lr)&& r2 = ? // int& && ==> int&  
10    decltype(rr)& r3 = ? // int&& & ==> int&  
11    decltype(rr)&& r4 = ? // int&& && ==> int&& 모두 두개씩 있을 때만 rvalue refernce로 인식!  
12 }
```

레퍼런스를 가리키는 레퍼런스는 명시적으로 코딩이 불가능하다. 하지만 `decltype()`을 통해 타입을 추론하게 되면 가능하다.

`decltype(&&)&&` 인 경우에만 `int&&` 타입으로 추론하고 나머지 경우는 `int&`로 추론한다. 이런 추론 규칙을 **reference collapsing**이라고 한다! reference collapsing은 **typedef, using, decltype, template** 4가지 경우에 적용된다.

```
1  template<typename T> void foo(T&& arg) { }
2
3  int main(){
4      int n=3;
5
6      typedef int& lref;
7      lref&& r1 = n;      // int& && ==> int&
8
9      using rref = int&&;
10     rref&& r2 = 10;    // int&& && ==> int&&
11
12     decltype(r2)&& r3 = 10; // int&& && ==> int&&
13
14     foo<int&&>(n);      // foo(int& && arg)
15     // foo(int& arg) 함수 생성!
16 }
```

2.12.4 Forwarding reference

```
1  void f1(int& arg) {}
2  void f2(int&& arg) {}
3  template<typename T> void f3(T& arg) {}
4
5  int main(){
6      int n=3;
7
8      f1(n); // ok
9      f1(0); // error
10
11     f2(n); // error
12     f2(0); // ok
13
14     f3(n); // ok. T는 어떤 값으로 받아도 lvalue reference이다.
15     f3(0); // error
16 }
```

`T&`는 어떤 타입으로 받아도 (`int&`, `int&&`) lvalue reference이다! 템플릿 케이스에 대해 보다 자세히 알아보자

```
1  template<typename T> void f3(T&& arg) \{\}
2
3  int main()\{
4      int n=3;
5
6      // 사용자가 명시적으로 타입을 추론한 경우
7      f3<int>(n);
8      f3<int&>(n);
9      f3<int&&>(n); // 3가지 케이스 모두 int&로 추론되므로 lvalue만 올 수 있다!
10
11     f3(n); // ok. 사용자가 템플릿 인자를 전달하지 않으면 int로 추론한다
12     f3(0); // error
13 \}
```

다음으로 템플릿 인자에 `T&&`가 붙어있는 경우를 생각해보자.

```
1  template<typename T> void f4(T&& arg) {}
2
3  int main()\{
4      int n=3;
5
6      f4<int>(n); // int &&. ==> int&& rvalue reference
```

```

7     f4<int&>(n);    // int& && ==> int&. 이 케이스에서만 lvalue reference가 된다!
8     f4<int&&>(n);   // int&& && ==> int&& rvalue reference
9
10    f4(n); // ok. 컴파일러가 자동으로 int 타입으로 변환해서 넘겨준다 (int ==> int)
11    f4(0); // ok.
12 }
```

T&& 와 같이 작성하면 타입 추론 규칙에 따라 rvalue와 lvalue를 같이 전달할 수 있다! 헷갈리기 쉬운 것이 함수가 하나인데 두 인자를 받는것이 아니라 함수가 2개가 생성되서 각각 따로 받는다. 그리고 생성된 각 함수는 call-by-value가 아닌 call-by-reference를 사용해서 전달받는다. 이러한 T&& reference를 **forwarding(universal) reference**라고 부른다!

2.13 Move semantics

2.13.1 Move constructor

다음과 같이 얕은 복사(shallow copy)가 일어나는 경우를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10 }
11
12 int main(){
13     Person p1("john", 20);
14     Person p2 = p1; // 얕은 복사 발생!
15 }
```

복사 생성자를 명시적으로 정해주지 않으면 컴파일러가 모든 멤버 변수 함수를 얕은복사하게 된다.

```

1 Person(const Person& p) : age(p.age) {
2     name = new char[strlen(p.name)+1];
3     strcpy_s(name, strlen(p.name)+1, p.name);
4 }
```

클래스 내에 다음과 같은 복사 생성자를 정의해주면 더 이상 얕은 복사가 발생하지 않고 깊은 복사가 발생한다! 하지만 복사 생성자는 성능 이슈가 존재한다. 다음과 같이 임시 객체를 반환하는 함수를 살펴보자

```

1 class Person{
2     char* name;
3     int age;
4     public:
5     Person(const char*s, int a) : age(a) {
6         name = new char[strlen(s)+1];
7         strcpy_s(name, strlen(s)+1, s);
8     }
9     ~Person() { delete[] name; }
10    Person(const Person& p) : age(p.age) {
11        name = new char[strlen(p.name)+1];
12        strcpy_s(name, strlen(p.name)+1, p.name);
13    }
14 }
15
16 Person foo() {
17     Person p("john", 20);
18     return p;
19 }
20
21 int main(){
22     Person ret = foo(); // 임시객체 반환하여 ret에 복사 생성자를 호출하여 복사해주고 바로 파괴됨!
```

}

`foo()` 함수는 값을 반환하므로 임시 객체가 생성되어 `ret`에 복사 생성자를 통해 깊은 복사를 일으키고 바로 파괴된다. `foo()`의 임시 객체를 파괴하지 않고 `ret`에 임시 객체의 주소를 그대로 가리킨 뒤 임시 객체의 메모리를 0으로 만드는 방법이 효율적이다!

```
1 Person(Person&& p) : name(p.name), age(p.age) {
2     p.name = nullptr;
3 }
```

Person 클래스 내부에 위와 같이 `move` 생성자를 만들면 rvalue만 받을 수 있고 임시 객체들은 해당 함수를 호출한다. 해당 코드가 rvalue를 처리하는 경우 기존의 복사 생성자보다 메모리 효율적이다!

2.13.2 std::move

```
1 class Object{
2     Object() = default;
3     Object(const Object& obj) { std::cout << "copy ctor" << std::endl; }
4     Object(Object&& obj) { std::cout << "move ctor" << std::endl; }
5 }
6
7 Object foo() {
8     Object obj;
9     return obj;
10 }
11
12 int main(){
13     Object obj1;
14     Object obj2 = obj1; // copy
15     Object obj3 = foo(); // move
16     Object obj4 = static_cast<Object&&>(obj1); // move
17     Object obj5 = std::move(obj2);           // move. 위 긴 변환문을 간단하게 move 함수를 호출하여 해결할
18     수 있다.
```

`std::move` 함수를 호출하면 `obj1, obj2` 같은 lvalue도 rvalue로 취급하여 `move` 생성자를 호출할 수 있다! `obj1, obj2`를 코드 내에서 더 이상 사용하지 않을 때 복사 생성자를 호출하기 보다 `move` 생성자를 호출하여 보다 효율적으로 값을 전달할 수 있다.

2.13.3 Move and noexcept

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::println("copy"); }
5     Object(Object&&) { std::println("move"); }
6 };
7
8 int main() {
9     std::vector<Object> v(3);
10    std::println("-----");
11    v.resize(5);           // copy, copy, copy 발생
12    std::println("-----");
13 }
```

`v(3)`로 처음 3개의 Object 자원을 확보한 후 `resize`를 통해 5개의 자원을 다시 확보한다고 5개 메모리를 새로 할당하고 기존 3개의 자원은 “복사(copy)”되어 새로운 메모리에 오게된다.

이렇게 `vector`의 베퍼를 새롭게 할당한 경우 결국 기존 베퍼를 제거하게 되므로 “복사(copy)”보다 “이동(move)”가 효율적이다! 하지만 위 코드를 실행하면 “복사(copy)”가 수행된다. 컴파일러가 기본적으로 복사를 수행하는 이유는 만약 이동을 하다가 예외가 발생하면 `vector`를 `resize` 이전 상태로 되돌릴 수 없다는 단점이 존재하기 때문이다.

따라서 이동을 사용하고 싶다면 되도록 예외가 발생하지 않도록 구현하고 `noexcept` 키워드를 붙여서 예외가 없음을

컴파일러에게 알려야 한다

```
1 class Object {
2     public:
3     Object() = default;
4     Object(const Object&) { std::println("copy"); }
5     Object(Object&&) noexcept { std::println("move"); } // 예외가 없음을 컴파일러에게 알림!
6 };
7
8 int main() {
9     Object o1;
10    Object o2 = o1;                                // copy
11    Object o3 = std::move(o1);                      // move
12    Object o4 = std::move_if_noexcept(o2);          // move
13
14    std::vector<Object> v(3);
15    std::println("-----");
16    v.resize(5);                                    // move, move, move 발생!
17    std::println("-----");
18 }
```

`std::move_if_noexcept`를 사용하면 컴파일러가 함수의 `noexcept` 키워드 유무를 검사하고 만약 키워드가 있다면 `move`를 실행한다. `std::move_if_noexcept`는 `type_traits` 기술로 예외 가능성을 조사한 후 예외 가능성이 있으면 `const T&` 타입으로 변환하고 예외 가능성이 없다면 `T&&` 타입으로 변환해주는 함수이다!

```
1 template<typename T>
2 constexpr std::conditional_t<
3     !std::is_nothrow_move_constructible_v<T> &&
4     std::is_copy_constructible_v<T>, const T&, T&&>
5     move_if_noexcept(T& x) noexcept {
6         return std::move(x)
7     }
```

다음과 같이 클래스가 여러 멤버 변수를 가지고 있는 경우를 살펴보자

```
1 template<typename T>
2 class Object{
3     int n;
4     std::string s;
5     T t;
6
7     public:
8     Object() = default;
9     Object(const Object& other) : n(other.n), s(other.s), t(other.t) {}
10    Object(Object&& other) noexcept
11        : n(other.n),
12        s(std::move(other.s)),
13        t(std::move(other.t))
14    {}
15};
```

복사 생성자를 보면 `int n`은 예외가 없다. 그리고 `std::string s`은 예외가 없음을 보장한다. 하지만 **임의의 타입 T는 예외가 존재할 가능성이 존재한다.** 이럴 땐 다음과 같이 키워드를 입력하면 된다.

```
1 Object(Object&& other) noexcept( noexcept( t(std::move(other.t)) ))
2     : n(other.n),
3     s(std::move(other.s)),
4     t(std::move(other.t))
5 {}
```

c++에서 `noexcept - 128) * 64 + (' - 128)` 꿀 두 가지 의미가 존재한다. 1) `noexcept operator`, 2) `noexcept specifier`

1. `noexcept operator`:

- `bool b = noexcept(expression)`: 어떤 표현식이 예외 가능성이 있는지 조사한다

2. noexcept specifier:

- `f() noexcept`, `f() noexcept(true)`: 함수 f는 예외가 없다.
- `f() noexcept(false)`: 함수 f는 예외가 존재한다

또 다른 방법으로는 다음과 같이 쓸 수 있다.

```
1 Object(Object&& other) noexcept( std::is_nothrow_move_constructible_v<T> )
2   : n(other.n),
3     s(std::move(other.s)),
4     t(std::move(other.t))
5   {}
```

2.13.4 Default move constructor

`move` 생성자를 기본적으로 컴파일러가 제공하는 경우에 대해 살펴보자

```
1 class Object{
2   std::string name;
3   public:
4     Object() = default;
5     Object(const Object& other) : name(obj.name) {}           // [1] 복사 생성자
6     Object& operator=(const Object& obj) { name = obj.name; } // [2] 복사 대입 연산자
7     Object(Object&& obj) : name(std::move(obj.name)) {}      // [3] move 생성자
8     Object& operator=(Object&& obj) { name = std::move(obj.name); } // [4] move 대입 연산자
9   };
```

1. **case 1.** 사용자가 [1,2,3,4] 모두 제공하지 않는 경우 컴파일러가 [1,2,3,4] 대한 default 버전을 제공한다.
2. **case 2.** 사용자가 [1] (또는 [2])만 제공하는 경우 컴파일러는 [2](또는 [1])는 default 버전을 제공하지만 [3,4]는 제공하지 않는다. 사실 [1]만 제공하면 컴파일러가 [2]도 제공하지 않는 것이 맞지만 설계 상 오류로 [2]는 default 버전이 제공된다. (since c++98)
3. **case 3.** 사용자가 [3](또는 [4])를 제공하는 경우 컴파일러는 [1], [2]를 삭제해버린다. 즉, 사용할 수 없다. 그리고 [4](또는 [3])는 제공하지 않는다.

하지만 코딩을 하다보면 복사 생성자는 사용자가 제공하지만 move 계열 함수만 컴파일러에게 요청하고 싶다.
이런 경우에는 어떻게 해야할까?

```
1 class Object{
2   std::string name;
3   public:
4     Object() = default;
5     Object(const Object& other) = default;      // 복사 대입 연산자도 move 생성자를 default로 요청하면
6     // 반드시 같이 요청해야 한다.
7     Object(Object&& obj) = default;
8     Object& operator=(Object&& obj) = default; // default 버전을 요청한다!
9   };
```

(...)=`default`;로 default 버전을 요청하면 컴파일러가 move 생성자는 `default` 버전을 생성한다. 복사 대입 연산자도 move 생성자를 `default`로 요청할 때 같이 요청해야 한다! 위와 같은 코드 형태는 잘 작성된 오픈소스에서 많이 볼 수 있다!

2.13.5 Rule of 3/5/0

다음으로 널리 통용되는 규칙인 Rule of 3/5/0에 대해 알아보자

```
1 class Person {
2   char* name;
3   int age;
4   public:
5     Person(const char*s, int a) :age(a) {
6       name = new char[strlen(s)+1];
7       strcpy(name, strlen(s)+1, s);
```

```

8   }
9   // char*와 같이 포인터 멤버변수가 있고 동적으로 메모리를 할당한다면
10  // c++98 시절에는 소멸자/복사 생성자/복사 대입연산자를 반드시 만들어야 했다 (Rule of 3)!
11  // c++11 시절에는 소멸자/복사 생성자/복사 대입연산자/ move 생성자/move 대입연산자 또한 만들어야 한다.
12  // (Rule of 5)
13 };

```

클래스 내부 변수에 포인터 멤버변수가 있고 동적으로 메모리를 할당한다면 반드시 선언해야 하는 함수를 가르켜 **Rule of 3 (c++98)**, **Rule of 5 (c++11)**라고 하였다.

`char*` 대신 `std::string`을 사용하면 사용자가 직접 자원을 관리할 필요가 없다. (= **Rule of 0**). 즉, STL에서 제공하는 클래스를 사용하면 클래스가 동적 메모리 할당에 대하여 컴파일러가 알아서 자동으로 제공한다. 결론은 STL을 많이 쓰면 좋다는 얘기이다.

2.14 Perfect forwarding

```

1 void foo(int n) {}
2 void goo(int& r) {r = 20; }
3
4 template<class F, class T>
5 void chronometry(F f, T arg) {
6     f(arg);
7 }
8
9 int main() {
10    int n = 10;
11    chronometry(foo, 10);
12    chronometry(goo, n); // error. T arg에 의해 값이 chronometry에 복사되어 goo에서 값을 20으로 바꿔도 n
13    // 값이 바뀌지 않는다.
14    std::cout << n << std::endl;
}

```

함수의 성능을 측정해주는 `chronometry` 템플릿 함수를 정의해보자. 이를 통해 두 함수 `foo`, `goo`를 하나의 함수 템플릿에서 처리하고 싶다. 이를 위해서는 perfect forwarding 기술이 필요하다. **Perfect forwarding**이란 전달 받은 인자를 다른 함수에게 “값, `const` 속성, value category 등의 변화없이 완벽하게 전달”하는 것을 말한다.

rvalue `n`과 lvalue `n`을 동시에 처리하기 위해 `const T&`를 사용할 수 있다. 하지만 이는 **전달 과정에서 const 속성이 추가되므로 완벽한 전달이라고 할 수 없다.**

```

1 void foo(const int n) {}
2 void goo(const int& r) { r=20; }
3
4 template<class F, class T>
5 void chronometry(F f, const T& arg) {
6     f(arg);
7 }

```

설명의 편의를 위해 우선 `T`를 `integer`로 고정해보자. lvalue와 rvalue를 속성 변화없이 그대로 받기 위해서는 다음과 같이 두 함수 템플릿이 필요하다.

```

1 template<class F, class T>
2 void chronometry(F f, int& arg) { // lvalue reference
3     f(arg);
4 }
5
6 template<class F, class T>
7 void chronometry(F f, int&& arg) { // rvalue reference
8     f(arg);
9 }

```

하지만 두 함수 템플릿은 `hoo()` 함수를 처리할 수 없다.

```

1 void hoo(int &&r) { }
2
3 template<class F, class T>
4 void chronometry(F f, int&& arg) { // rvalue reference
5     f(arg);
6 }

```

```

6   }
7
8   int main() {
9     hoo(10); // ok
10    chronometry(hoo, 10); // error
11 }
```

chronometry에서 rvalue reference `int&&`로 받아도 `arg`라는 이름이 생기기 때문에 더 이상 rvalue가 아니게 된다. 즉, 10은 rvalue이지만 `arg`는 lvalue이다.

```

1 template<class F, class T>
2 void chronometry(F f, int&& arg) { // rvalue reference
3   // f(arg);
4   f(static_cast<int&&>(arg)); // 다시 rvalue로 캐스팅해서 보내야 함!
5 }
```

따라서 rvalue를 호출할 때는 `static_cast`로 다시 한 번 rvalue로 캐스팅해서 넘겨줘야 한다.

2.14.1 Using forwarding reference

Forwarding reference를 사용하면 `int&`, `int&&`를 자동 생성할 수 있다. 따라서 함수 템플릿을 두 개 생성하지 않아도 된다. 하지만 템플릿을 사용하려면 구현부가 동일해야 하는데 하나는 캐스팅이 있고 하나는 캐스팅이 없다.

```

1 void chronometry(F f, int& arg) { // lvalue reference
2   f(static_cast<int&>(arg));
3 }
4
5 template<class F, class T>
6 void chronometry(F f, int&& arg) { // rvalue reference
7   f(static_cast<int&&>(arg));
8 }
```

따라서 lvalue에도 캐스팅을 추가해준다. 컴파일 타임에서 lvalue 캐스팅은 같은 타입 캐스팅이기 때문에 무시된다. 이제 구현부까지 동일해졌으므로 하나의 템플릿 함수로 합칠 수 있다.

```

1 template<class F, class T>
2 void chronometry(F f, T&& arg) { // rvalue reference
3   f(static_cast<T&&>(arg));
4 }
5
6 int main() {
7   chronometry(goo, n); // T = int&, T&& = int&
8   chronometry(hoo, 10); // T = int, T&& = int&&
9 }
```

Perfect forwarding은 `const`까지 자동으로 커버해주기 때문에 `const` 전용 함수를 만들지 않아도 된다. C++ 표준에는 캐스팅을 대신 해주는 함수가 존재한다. `static_cast`를 사용하는 대신 `std::forward` 함수를 사용한다.

```

1 template<class F, class T>
2 void chronometry(F f, T&& arg) { // rvalue reference
3   f( std::forward<T>(arg) );
4 }
```

`std::forward` 함수는 forward referencing을 자동으로 해주는 함수이며 lvalue를 (함수로 전달하면) lvalue로 캐스팅하고 rvalue를 (함수로 전달하면 받으면서 lvalue로 변경된 것을 다시) rvalue로 캐스팅해준다. 즉, 인자가 rvalue일 때만 `std::move()`를 하는 것을 의미한다!

2.14.2 Variadic parameter template

만약 `foo`와 `goo`의 파라미터 개수가 다르다면 어떻게 해야 할까? template에서 가변인자를 받는 `...`을 사용한다.

```

1 void foo() {}
2 int& goo(int a, int& b, int&& c) {
3   b=20;
```

```

4     return b;
5 }
6
7 template<class F, class ... T>
8 void chronometry(F f, T&& ... arg) { // rvalue reference
9     f( std::forward<T>(arg) ... );
10 }
```

goo는 리턴값이 존재한다. 이런 경우 어떻게 해야할까? `auto`를 사용하게 되면 참조값을 버리는 특성이 있으므로 `decltype(auto)`를 사용한다.

```

1 template<class F, class ... T>
2 decltype(auto) chronometry(F f, T&& ... arg) { // rvalue reference
3     return f( std::forward<T>(arg) ... );
4 }
```

Perfect forwarding을 정리하면 다음과 같다

1. 인자를 받을 때 **forwarding reference (`T&&`)**를 사용한다.
2. 인자를 다른 함수에 전달할 때 `std::forward<T>(arg)`로 묶어서 전달한다.
3. 여러 개의 인자를 모두 forwaring하기 위해 **가변인자 템플릿**을 사용한다.
4. 반환 값도 전달하기 위해서는 `decltype(auto)`로 반환한다.

2.14.3 Member function pointer for perfect forwarding

chronometry 함수에 멤버 함수를 전달할 수는 없을까?

```

1 void foo(int n) {}
2
3 class Test{
4     public:
5         void f1(int n) { cout << "Test t1" << endl; }
6     };
7
8 template<class F, class ... T>
9 decltype(auto) chronometry(F f, T&& ... arg){
10     return f(std::forward<T>(arg)...);
11 }
12
13 int main() {
14     chronometry(foo, 10); // ok
15
16     Test obj;
17     chronometry(&Test::f1, &obj, 10); // error. 이걸 가능하게 할 순 없을까?
18 }
```

`std::invoke`를 사용하면 일반함수 포인터는 `std::invoke(f, arg)`와 같이 호출하고 멤버함수 포인터는 `std::invoke(f, &obj, arg)`와 같이 호출한다.

```

1 void foo(int n) {}
2
3 class Test{
4     public:
5         void f1(int n) { cout << "Test t1" << endl; }
6     };
7
8 template<class F, class ... T>
9 decltype(auto) chronometry(F f, T&& ... arg){
10     return std::invoke(f, std::forward<T>(arg)...);
11 }
12
13 int main() {
14     chronometry(foo, 10); // ok
15 }
```

```

16     Test obj;
17     chronometry(&Test::f1, &obj, 10); // ok.
18 }
```

2.14.4 Function object for perfect forwarding

함수 객체 functor에 () 대입연산자를 재정의하면 함수처럼 쓸 수 있다.

```

1 struct Functor {
2     void operator()(int n) & { // lvalue 객체일 때 실행
3         cout << "operator()" &"<< endl;
4     }
5
6     void operator()(int n) && { // rvalue 객체일 때 실행
7         cout << "operator() &&"<< endl;
8     }
9 }
10
11 template<class F, class ... T>
12 decltype(auto) chronometry(F f, T&& ... arg){
13     return std::invoke(f, std::forward<T>(arg)...);
14 }
15
16 int main() {
17     Functor f;
18
19     chronometry(f, 10);           // ok. lvalue operator 정상적으로 호출
20     chronometry(Functor(), 10); // error. rvalue operator 가 호출되지 않고 lvalue opeartor가 호출됨
21     // (전달받을 때 복사본을 만들기 때문)
}
```

f를 넘겨줄 때 (F f)로 받고 있기 때문에 Functor()와 같이 rvalue를 호출해도 lvalue로 변환되는 것이다. 따라서 함수도 진짜 함수가 아닌 함수 객체가 올 수 있는데 이럴 경우 F 또한 forwarding reference(&&)로 넘겨줘야 한다. 그리고 받을 때도 std::forward<F>로 받아야 한다.

```

1 struct Functor {
2     void operator()(int n) & { // lvalue 객체일 때 실행
3         cout << "operator()" &"<< endl;
4     }
5
6     void operator()(int n) && { // rvalue 객체일 때 실행
7         cout << "operator() &&"<< endl;
8     }
9 }
10
11 template<class F, class ... T>
12 decltype(auto) chronometry(F&& f, T&& ... arg){
13     return std::invoke(std::forward<F> f, std::forward<T>(arg)...);
14 }
15
16 int main() {
17     Functor f;
18
19     chronometry(f, 10);           // ok
20     chronometry(Functor(), 10); // ok
21 }
```

2.14.5 Chronometry implementation

함수가 돌아가는데 걸리는 시간을 측정하기 위해 StopWatch 클래스를 작성해준다.

```

1 class StopWatch{ // in StopWatch.h
2 public:
3     StopWatch() : start(chrono::system_clock::now()) {}
```

```

4
5     ~StopWatch() {
6         end = chrono::system_clock::now();
7
8         chrono::duration<double> elapsed = end - start;
9
10        cout << elapsed.count() << " seconds..." << endl;
11    }
12
13    private:
14        chrono::system_clock::time_point start;
15        chrono::system_clock::time_point end;
16    };

```

chronometry 함수 내 한 줄만 추가해주면 시간을 측정하는 코드를 작성할 수 있다.

```

1 #include "StopWatch.h"
2
3 void foo(int n) {
4     std::this_thread::sleep_for(std::chrono::seconds(n));
5 }
6
7 template<class F, class ... T>
8 decltype(auto) chronometry(F&& f, T&& ... arg){
9     StopWatch sw; // 한 줄만 적으면 시간을 측정해준다.
10    return std::invoke(std::forward<F> f, std::forward<T>(arg)...);
11 }
12
13 int main() {
14     chronometry(foo, 2);
15 }

```

2.14.6 Notice for perfect forwarding

Perfect forwarding으로 인자를 넘겨줄 때 암시적인 중괄호 전달 대신 명시적으로 전달해야 한다.

```

1 void foo(std::pair<int, int> p) {}
2
3 int main() {
4     chronometry(foo, {1,2}); // error
5     chronometry(foo, std::pair{1,2}); // ok
6 }

```

일반 foo 함수를 호출할 때 foo(1,2)는 정상적으로 동작하지만 perfect forwarding에서는 명시적으로 타입을 정의해줘야 한다.

같은 이름의 함수가 오버로딩된 경우 perfect forwarding 시 함수의 모양을 캐스팅해줘야 한다.

```

1 void goo(int a) {}
2 void goo(int a, int b) {}
3
4 int main(){
5     chronometry(goo, 1, 2); // error. goo를 넘겨주는 시점에서 어떤 함수를 넘겨줘야 할지 모른다.
6
7     chronometry(static_cast<void(*)(int, int)>(goo), 1,2); // ok
8 }

```

2.14.7 Perfect forwarding in STL

이번 섹션에서는 perfect forwarding 기술이 stl에서 어떻게 사용되고 있는지 살펴본다.

```

1 class Point{
2     int x,y;
3     public:
4     Point(int x, int y) :x(x), y(y) {}
5     Point(const Point& pt) :x(pt.x), y(pt.y) {}

```

```

6     ~Point() {}
7 };
8
9 int main() {
10    vector<Point> v;
11
12    Point pt(1,2);
13    v.push_back(pt); // 이 순간 어떤 일이 발생할 것인가?
14
15    cout << "-----" << endl;
16 }

```

v.push_back(pt)가 호출되는 순간 v 내부 버퍼에 사용자가 전달한 Point 객체의 복사본을 생성한다. 즉, 12번 줄에서 Point 객체를 한 번 생성하고 13번 줄에서 push_back이 실행되면서 Point 객체가 복사로 인해 한 번 더 생성된다.

```

1 int main() {
2    vector<Point> v;
3
4    Point pt(1,2); // Point(int, int) 호출
5    v.push_back(pt); // Point(const Point&) 호출
6
7    cout << "-----" << endl;
8    // 소멸자 2개 호출
9 }

```

다른 방법으로 임시객체를 만들어보자.

```

1 int main() {
2    vector<Point> v;
3
4    v.push_back(Point(1,2)); // 임시 객체 호출하면 소멸자가 불리는 시점만 달라질 뿐 2개가 생성되는 것은
5    // 동일
6    // Point 임시 객체는 바로 파괴됨
7
8    cout << "-----" << endl;
9    // v에 들어간 Point 객체의 소멸자 호출
}

```

좀 더 효율적인 방법은 없을까? `emplace_back`을 사용하면 객체 자체를 전달하지 않고 객체를 생성할 때 필요한 데이터만 전달할 수 있다.

```

1 int main() {
2    vector<Point> v;
3
4    v.emplace_back(1,2); // 생성자 1개만 호출
5
6    cout << "-----" << endl;
7    // v에 들어간 Point 객체의 소멸자 호출
8 }

```

```

1 template<class ... Args>
2 constexpr reference emplace_back(Args&& ... args); // since c++20 (from cppreference.com)

```

Type이 Point로 변환되고 args에 (1,2)가 전달되어 호출된다. 만약 생성자가 lvalue reference 또는 rvalue reference를 입력으로 받아도 `emplace_back`은 정상적으로 동작해야 한다. 이를 위해서 `emplace_back`는 **perfect forwarding 기술**을 사용하여 설계되었다.

```

1 template<typename ... Ts>
2 decltype(auto) emplace_back(Ts&& ... args) {
3     Type* t = new Type(std::forward<Ts>(args) ...);
4     return *t;
5 }

```

따라서 stl 컨테이너에서 사용자 정의 타입을 값(value)로 보관할 때는 push 계열의 함수 대신 emplace 계열의

함수를 사용하는 것이 좋다.

2.15 Callable object

2.15.1 Function object

```
1 struct plus{
2     int operator()(int arg1, int arg2) { return arg1 + arg2; }
3 };
4
5 int main() {
6     plus p;
7     int n = p(1,2);      // p는 객체이지만 함수처럼 사용하고 있다.
8     cout << n << endl;
9 }
```

함수 객체란 괄호 ()를 사용해서 호출 가능한 객체를 말한다. a와 b가 객체일 때 두 객체 사이의 연산은 컴파일러가 아래와 같이 변환해준다. 이를 **연산자 재정의**라고 한다.

```
1 a + b    // a.operator+(b)
2 a - b    // a.operator-(b)
3 a()       // a.operator()()
4 a(1,2)   // a.operator()(1,2)
```

함수 대신 함수 객체를 사용하는 이유가 뭘까?

1. 함수와 달리 상태를 가질 수 있다.
2. 특정 상황에서 함수보다 빠르게 사용할 수 있다(인라인 치환).
3. 모든 함수 객체는 자신만의 타입을 가진다.

앞서 정의한 plus 객체를 완성도 있게 만들어보자.

```
1 template<class T>
2 struct plus {
3     [[nodiscard]] constexpr
4     T operator()(const T& arg1, const T& arg2) const {
5         return arg1 + arg2;
6     }
7 };
8
9 int main() {
10     const plus<int> p; // template으로 다양한 타입들을 사용할 수 있고 const plus 객체 또한 const 구문을
11     // 추가함으로써 사용할 수 있다.
12     p(3,4);           // [[nodiscard]]에 의해 컴파일 경고문이 뜬다.
13     int n = p(1,2);
14     cout << n << endl;
15 }
```

함수 객체를 작성할 때 권장되는 가이드는 다음과 같다.

1. 템플릿으로 만들어서 다양한 타입들이 호환 가능하도록 할 것
2. `operator()`는 `const` 멤버 함수로 작성할 것
3. `constexpr`, `[[nodiscard]]`를 사용하여 컨벤션을 강화할 것
4. `noexcept`이 필요한 경우 사용할 것
5. perfect forwarding & template specialization 기술을 적용할 것
6. `is_transparent` 멤버 함수가 필요하면 추가할 것

2.15.2 Function object = function with state

함수가 상태를 가진다는 말은 무엇을 의미할까? 다음 예제를 보자.

```
1 int urand() {
2     return rand() % 10;
3 }
4
5 int main() {
6     for(int i=0; i<10; i++){
7         cout << urand() << ", ";
8     }
9     cout << endl;
10 }
```

위 함수를 실행하면 난수가 생성된다. 이 때 중복되는 난수가 생성될 수 있는데 만약 중복되지 않은 난수를 구하려면 어떻게 해야 할까? 그러기 위해서는 한 번 반복했던 난수 값은 어딘가에 보관되어야 한다. 하지만 함수는 '동작은 있으나 상태는 없으므로' 한 번 반복한 값은 잊어버리고 다시 새롭게 동작한다. 반면에 **함수 객체를 사용하면 함수의 난수 값을 상태 변수에 기억하여 중복되지 않게 호출할 수 있다.**

```
1 class URandom {
2     // member data에 저장하여 상태를 기억할 수 있다!
3     public:
4         int operator()() {
5             return rand() % 10;
6         }
7     };
8
9     int main () {
10         URandom urand;
11         for(int i=0; i<10; i++){
12             cout << urand() << ", ";
13         }
14         cout << endl;
15     }
```

클래스를 완성도 있게 작성해보자

```
1 class URandom{
2     std::bitset<10> bs; // 10-bit을 관리하기 위한 멤버 변수
3     bool recycle;
4
5     std::mt19937 rancgen { std::random_device{}() };
6     std::uniform_int_distribution<int> dist{0, 9};
7     public:
8     URandom(bool recycle = false) : recycle(recycle) {
9         // bs.set(5); // 5번째 비트만 1로
10        bs.set(); // 모든 비트를 1로
11    }
12
13    int operator()(){
14        if(bs.none()) { // 모든 값을 꺼낸 경우 recycle 변수에 따라 다시 순행하거나 -1 리턴
15            if(recycle) bs.set();
16            else return -1;
17        }
18
19        int k=-1;
20        while( !bs.test( k = dist(rancgen) )); // 난수를 구하고 그 값이 bs에 1로 세팅되어 있는지 확인. 0
21        // 이면 이미 나온 값이므로 다른 값을 난수로 구함
22        bs.reset(k); // 처음 나온 값은 0으로 세팅
23
24        return k;
25    }
26};
```

2.15.3 closure & function object

```
1  bool f1(int a) { return a%3 == 0; }
2
3  int main() {
4      std::vector<int> v = { 1,2,6,7,8,3,4,5,9,10};
5      auto r1 = std::find(v.begin(), v.end(), 3);
6      auto r2 = std::find_if(v.begin(), v.end(), f1);
7  }
```

std::find는 주어진 컨테이너에서 값을 바로 찾는 함수이고 std::find_if는 주어진 컨테이너에서 조건자(predicate)에 맞는 값을 찾는 함수이다. 조건자(predicate)란 bool(또는 bool로 변환 가능한 타입)을 반환하는 함수 또는 함수 객체를 말한다.

위 코드에서 f1은 3의 배수를 찾는 함수였다. 만약 사용자로부터 입력받는 정수 k의 배수를 검색하게 하고 싶다면 어떻게 해야 할까? 전역 변수를 쓰면 해결할 수 있지만 이런 코드를 작성할 때마다 전역 변수를 생성한다면 매우 비효율적인 코딩이 된다. 함수 객체를 사용하면 이를 쉽게 해결할 수 있다.

```
1  class F{
2      int value;
3  public:
4      F(int v) : value(v) {}
5
6      bool operator()(int n) const {
7          return n % value == 0;
8      }
9  };
10
11 int main() {
12     std::vector<int> v = { 1,2,6,7,8,3,4,5,9,10};
13     int k = 3;
14     auto r1 = std::find_if(v.begin(), v.end(), F(k)); // 함수 객체를 사용하여 사용자로부터 입력받은 k의
15     배수를 쉽게 구할 수 있다.
16     cout << *r1 << endl;
}
```

위와 같이 함수 객체는 scope 내에 지역 변수를 캡처할 수 있는 기능이 있는데 이를 closure라고 한다. 프로그래밍 분야에서 closure의 자세한 정의는 scope 내의 지역 변수를 바인딩 할 수 있는 일급함수객체(first-class object)이고 1960년대 처음 용어가 등장하였다.

2.15.4 inline & function object

인라인.inline) 함수는 컴파일 시에 함수 호출식을 함수의 기계어 코드로 치환하는 것을 의미한다.

```
1  int add1(int a, int b) { return a + b; }
2  inline int add2(int a, int b) { return a + b; }
3
4  int main () {
5      int ret1 = add1(1,2); // 함수 호출
6      int ret2 = add2(1,2); // 치환
7
8      int(*f)(int, int) = &add2; // add2를 함수 포인터로 사용하면 호출될까?
9      f(1, 2);
10 }
```

인라인 함수는 컴파일 타임에 치환되기 때문에 함수 포인터 (*f)를 통해 인라인 함수 add2를 가리키면 치환되지 않고 호출된다. (*f)는 런타임 시 수시로 가리키는 포인터가 달라질 수 있으므로 예측하기 어렵기 때문이다.

다른 예제를 보자. sort 함수의 비교 정책을 함수 포인터로 바꾸고 싶은데 속도 향상을 위해 cmp를 인라인 함수로 치환하면 정말 속도가 빨라질까?

```
1  void sort(int* x, int sz, bool(*cmp)(int, int)) { // 함수 포인터를 사용하여 비교 정책 변경
2      for(int i=0; i<sz-1; i++){
3          for(int j=i+1; j<sz; j++){
4              if(x[i] > x[j])
5                  if(cmp(x[i], x[j]))
6                      std::swap(x[i], x[j]);
7      }
8  }
```

```

7         }
8     }
9 }
10
11 inline bool cmp1(int a, int b) {return a<b;}
12 inline bool cmp2(int a, int b) {return a>b;}
13
14 int main() {
15     int x[10] = {1,3,5,7,9,2,4,6,8,10};
16     sort(x, 10, &cmp1);
17     sort(x, 10, &cmp2); // 인라인 치환이 되었을 거라고 예상했으나 함수 포인터로 인라인 함수를 호출하면
18     치환되지 않는다!
}

```

cmp1과 cmp2는 다른 함수지만 함수 포인터로 가리키게 되면 signature(반환 타입과 인자 모양)이 같으므로 동일한 함수 타입으로 간주된다. 즉, **함수는 자신만의 타입이 없다.** 반면에 함수 객체는 signature가 동일해도 클래스 이름으로 구분할 수 있다. 따라서 **함수 객체는 자신만의 타입이 있다.**

```

1 inline bool cmp1(int a, int b) {return a<b;}
2 inline bool cmp2(int a, int b) {return a>b;}
3
4 struct Less {
5     inline bool operator()(int a, int b) const { return a < b; }
6 };
7
8 struct Greater{
9     inline bool operator()(int a, int b) const { return a > b; }
10};
11
12 template<class T>
13 void sort(int* x, int sz, T cmp) {
14     for(int i=0; i<sz-1; i++){
15         for(int j=i+1; j<sz; j++){
16             if(cmp(x[i], x[j]))
17                 std::swap(x[i], x[j]);
18         }
19     }
20
21     int main(){
22         int x[10] = {1,3,5,7,9,2,4,6,8,10};
23         Less f1;
24         Greater f2;
25
26         sort(x, 10, &cmp1);
27         sort(x, 10, &cmp2);
28         sort(x, 10, f1);
29         sort(x, 10, f2);
30     }
31 }

```

sort 비교 정책으로 일반 함수(cmp1, cmp2)를 사용하면 signature가 동일하기 때문에 비교 정책을 교체해도 **코드 메모리가 증가하지 않는다.** 하지만 비교 정책 함수가 인라인 치환될 수 없기 때문에 느린다. 이와 반대로 sort 비교 정책으로 함수 객체(Less, Greater)를 사용하면 비교 정책이 인라인 치환되기 때문에 **빠르지만 정책을 교체한 횟수 만큼의 sort() 함수가 생성되어 코드 메모리가 증가한다.**

2.15.5 STL & function object

c++ 표준에는 이미 많은 함수 객체를 제공한다. 이는 functional 헤더에 포함되어 있다.

```

1 #include <functional>
2
3 int main() {
4     int x[10] = {1,3,5,7,9,2,4,6,8,10};
5
6     std::greater<int> f;
7     std::sort(x, x+10, f);

```

```
8     std::sort(x, x+10, std::greater<int>()); // 임시 객체 표기법
9     std::sort(x, x+10, std::greater());      // c++17부터 타입 생략 가능
10    std::sort(x, x+10, std::greater{});      // 중괄호 표기법 가능
11 }
```

3 STL programming

3.1 STL structure

3.1.1 Generic algorithm - find

문자열 내에서 원하는 문자를 찾는 strchr 함수를 작성해보자

```
1 char* mystrchr(char* s, int c) {
2     while(*s != c) {
3         if(!*s++) return nullptr;
4     }
5     return s;
6 }
7
8 int main() {
9     char s[] = "abcdefg";
10
11     char *p = mystrchr(s, 'c');
12
13     if(p == nullptr) std::println("fail");
14     else             std::println("success : {}", *p);
15 }
```

전체 문자열이 아니라 부분 문자열 검색을 가능하게 할 수는 없을까? 문자열 "시작 뿐 아니라 끝도 전달"해야 한다

- 방법1: 시작 주소와 요소의 개수 전달
- 방법2: 시작 주소와 마지막 주소를 전달

이 중 c++에서는 방법2를 채택하여 사용하고 있다. 구간의 끝(last)을 검색 대상에 포함할 것인가? 포함하지 않는 것이 장점이 많다

- s = [a,b,c,d,e,f,g]과 같이 주어졌을 때 first=s, last=s+4인 경우 [a,b,c,d]만이 문자열 구간에 포함된다. 이 때, s='a', s+4='e'이다.
- 이를 [first, last)와 같이 표기하며 반개행 구간(half open range)라고 부른다.

```
1 char* mystrchr(char* first, char* last, int c) {
2     for(; first != last; ++first) {
3         if(*first == c) return first;
4     }
5     return nullptr;
6 }
7
8 int main() {
9     char s[] = "abcdefg";
10
11     char *p = mystrchr(s, s+4, 'e');
12
13     if(p == nullptr) std::println("fail");
14     else             std::println("success : {}", *p);
15 }
```

- 위 코드에서는 s='a', s+4='e'를 가르키고 있으므로 e는 문자열 구간에 포함되지 않는다. 따라서 fail이 출력된다.

위 코드를 일반화하여 문자열 뿐만 아니라 모든 타입의 배열에서 선형 검색을 하는 함수를 작성해보자. 템플릿 (template) 프로그래밍을 하자는 의미이다.

```
1 template<typename T>
2 T* find(T* first, T* last, T c) { // 모든 배열에서 원소를 찾으므로 이름을 일반화하여 find로 명명한다
3     for(; first != last; ++first) {
4         if(*first == c) return first;
5     }
6     return nullptr;
```

```

7   }
8
9 int main() {
10    char s[] = "abcdefg";
11    char* p = find(s, s+4, 'e');           // char 뿐만 아니라
12
13    double d[] = {1,2,3,4,5,6,7,8,9,10};
14    double* pd = find(d, d+10, 5.0);      // double 타입도 작동!
15
16    if(p == nullptr) std::println("fail");
17    else            std::println("success : {}", *p);
18 }
```

- 혹자는 10개 크기의 배열을 만들고 11번째 주소를 레퍼런싱하는게 말이 되냐고 물을 수 있는데 c++ 표준문서에 보면 비교 연산을 위해 n개 크기의 배열의 n+1 주소를 사용하는 것은 문제없다고 되어 있다!

위 코드에는 다소 황당한 컴파일 에러가 존재한다.

```

1 template<typename T>
2 T* find(T* first, T* last, T c) {
3     for(; first != last; ++first) {
4         if(*first == c) return first;
5     }
6     return nullptr;
7 }
8
9 int main() {
10    double d[] = {1,2,3,4,5,6,7,8,9,10};
11    double* pd = find(d, d+10, 5);        // error! double형 타입이 아닌 int형 타입은 못찾음!
12    double* pd = find(d, d+10, 5.0f);    // error! double형 타입이 아닌 float형 타입은 못찾음!
13 }
```

- double 배열에서 int, float을 못찾는 것은 말이 안된다([치명적 단점](#)).

- 함수 인자로 T*로 표기하면 Raw Pointer 사용 가능하다. 이는 즉, 포인터처럼 작동하는 객체들(스마트 포인터, 반복자 등)을 사용할 수 없다는 의미.

- 해결책1: 구간을 나타내는 타입(T1)과 검색 대상 타입(T2)를 분리한다.
- 해결책2: T* 대신 T1으로 표기해서 구간을 나타내는 타입이 "[반드시 포인터이어야 한다는 조건을 제거](#)"한다. 단, 구간을 나타내는 타입은 ==, !=, ++, * 연산이 가능해야 한다.

```

1 template<typename T1, typename T2>
2 T1 find(T1 first, T1 last, const T2& value);
```

- T1 first, T1 last → 구간을 나타내는 타입으로 포인터 또는 반복자가 올 수 있다. 일반적으로 "call by value"로 받는다

- const T2 value → 검색할 요소. user type이 될 수도 있다. 복사본의 오버헤드를 줄이기 위해 일반적으로 "const reference"로 받는다

현재 코드는 검색 실패 시 nullptr를 반환한다. nullptr 대신 "last"를 반환하게 하면 어떨까? "last"는 검색 대상이 아니므로 검색 성공 시 last가 반환될 수는 없다. [first, last) 구간의 검색 실패 시 "[last는 다음 구간의 시작](#)"으로 사용될 수 있다.

```

1 template<typename T1, typename T2>
2 constexpr T1 find(T1 first, T1 last, const T2& value);
3 for(; first != last; ++first) {
4     if(*first == value) return first;
5 }
6 return last; // 검색 실패 시 nullptr이 아닌 last를 반환!
7 }
8
9 int main() {
10    double d[] = {1,2,3,4,5,6,7,8,9,10};
```

```

11     double* p = find(d, d+5, 7);
12
13     if(p == d+5) std::println("fail");
14     else         std::println("success : {}", *p);
15 }
```

- 함수 앞에 constexpr을 붙이면 컴파일 시간에 수행되어 성능이 좋아진다.

지금까지 작성한 find 함수는 c++ STL에 <algorithm> 헤더에 미리 구현되어 있다(c++98). 지금까지 구현한 find 함수처럼 템플릿 기반의 함수들을 "Generic Algorithm"이라고 부른다.

- 생각해 볼 문제 → 주어진 구간에서 "특정 값"이 아닌 "조건을 검색"하게 할 수 없을까? ex) [first, last) 구간에서 첫번째 나오는 짝수를 찾고 싶다.

3.1.2 Make iterator

다음과 같은 Single Linked List 코드를 보자.

```

1 template<typename T> struct Node {
2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
7 template<typename T> class slist {
8     Node<T>* head=nullptr;
9     public:
10    void push_front(const T& data) {
11        head = new Node<T>(data, head); // 아래와 같이 입력된다.
12        // 10, 0
13        // 20, 100
14        // 30, 200
15        // 40, 300
16        // 50, 400
17    }
18 };
19
20 int main() {
21     slist<int> s;
22     s.push_front(10);
23     s.push_front(20);
24     s.push_front(30);
25     s.push_front(40);
26     s.push_front(50);
27 }
```

- 주제에 집중하기 위해 "move 지원, 요소 제거" 등은 생략하고 push_front() 멤버 함수만 제공한다.

slist도 배열처럼 여러 개의 값을 보관하고 있다. slist에서 값을 검색하기 위해 앞서 만든 find() 알고리즘을 사용할 수 있을까? → 배열이 아니므로 불가능!

- 배열 - [50,40,30,20,10]
- 리스트 - [50, 0x400] → [40, 0x300] → [30, 0x200] → [20, 0x100] → [10, 0x0]

p,q가 각각 배열과 리스트의 첫번째 요소를 가리키는 포인터일 때

다음 요소로 이동	<code>++p</code>	<code>q = q->next</code>
요소에 접근	<code>*p</code>	<code>q->data</code>

위와 같이 두 자료구조는 이동, 접근 방식이 전부 다르다! 앞서 작성한 find() 알고리즘은 포인터에 맞춰 작성되어 있기 때문에 slist 리스트에는 사용할 수 없다.

일관성을 위해 리스트에서도 `++q`, `*q` 연산자가 동작해야 한다. 이를 위해 반복자(iterator)를 별도로 작성해야 한다.

```

1 template<typename T> struct Node {
2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
7 template<typename T> class slist_iterator {           // 반복자(iterator)
8     Node<T>* current;
9     public:
10    slist_iterator(Node<T>* p = nullptr) : current{p} {}
11
12    slist_iterator& operator++() {                   // ++p, *p, p==q, p!=q를 위한 연산자 재정의
13        current = current->next;
14        return *this;
15    }
16    T& operator*() {
17        return current->data;
18    }
19    bool operator==(const slist_iterator& it) const { return current == it.current; }
20    bool operator!=(const slist_iterator& it) const { return current != it.current; }
21 };
22
23 slist_iterator<int> p{0x500};
24 ++p;           // ok. 연산자 동작
25 int n = *p;   // ok. 연산자 동작

```

3.1.3 Generic Algorithm - list

이전 섹션에서 만든 코드를 find() 알고리즘과 섞어보자. find를 사용하여 slist의 모든 요소를 검색하려면 첫번째 요소를 가리키는 반복자와 "마지막 다음 요소"를 가리키는 반복자가 필요하다.

```

1 template<typename T> struct Node {
2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
7 template<typename T> class slist_iterator {           // 반복자(iterator)
8     Node<T>* current;
9     public:
10    slist_iterator(Node<T>* p = nullptr) : current{p} {}
11
12    slist_iterator& operator++() {                   // ++p, *p, p==q, p!=q를 위한 연산자 재정의
13        current = current->next;
14        return *this;
15    }
16    T& operator*() {
17        return current->data;
18    }
19    bool operator==(const slist_iterator& it) const { return current == it.current; }
20    bool operator!=(const slist_iterator& it) const { return current != it.current; }
21 };
22
23 template<typename T> class slist {
24     Node<T>* head=nullptr;
25     public:
26    void push_front(const T& data) {
27        head = new Node<T>(data, head);
28    }
29
30    slist_iterator<T> begin() { return slist_iterator<T>{head}; } // begin(), end() 함수 제공해야함!
31    slist_iterator<T> end() { return slist_iterator<T>{nullptr}; }
32 };
33

```

```

34 int main() {
35     slist<int> s;
36     s.push_front(10);
37     s.push_front(20);
38     s.push_front(30);
39     s.push_front(40);
40     s.push_front(50);
41
42     slist_iterator<int> p1 = s.begin(); // 반복자 받은 후
43     slist_iterator<int> p2 = s.end();
44
45     while(p1 != p2) {           // 모든 요소 순회
46         std::print("{} , ", *p1);
47         ++p1;
48     }
49 }
```

- STL 규칙 - 모든 컨테이터는 반드시 `begin()`, `end()` 멤버 함수가 제공되어야 한다.

컨테이너와 반복자의 클래스 이름

- 컨테이너 클래스의 이름은 C++ 표준에 의해서 정해져 있다. ex) `vector`, `queue`, `list`, ...
- 하지만 반복자의 클래스 이름은 정해져 있지 않다.

따라서 컨테이너 설계자가 지켜야 하는 규칙이 있다. 반드시 자신의 반복자 클래스 이름은 "iterator"라는 약속된 이름으로 외부에 노출하여야 한다.

```

1 template<typename T> class slist {
2     Node<T>* head=nullptr;
3
4     public:
5         using iterator = slist_iterator<T>;           // 반복자는 iterator라는 이름으로 사용할 수 있다.
6
7         void push_front(const T& data) {
8             head = new Node<T>(data, head);
9         }
10
11         iterator begin() { return iterator{head}; } // 임의의 이름 -> iterator로 변경!
12         iterator end() { return iterator{nullptr}; }
13     };
14
15     int main() {
16         slist<int> s;
17         s.push_front(10);
18         s.push_front(20);
19         s.push_front(30);
20         s.push_front(40);
21         s.push_front(50);
22
23         slist<int>::iterator p1 = s.begin(); // 반복자는 다음과 같이 호출한다! 일반적으로 auto를 사용하여 이를
24         // 생략할 수 있다.
25         slist<int>::iterator p2 = s.end();
26
27         while(p1 != p2) {
28             std::print("{} , ", *p1);
29             ++p1;
30         }
31 }
```

반복자까지 구현되어 있으면 `find()` 알고리즘이 정상적으로 동작한다.

```

1 template<typename T> struct Node {
2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data{d}, next{n} {}
5 };
6
```

```

7 template<typename T> class slist_iterator {
8     Node<T>* current;
9     public:
10    slist_iterator(Node<T>* p = nullptr) : current{p} {}
11
12    slist_iterator& operator++() {
13        current = current->next;
14        return *this;
15    }
16    T& operator*() {
17        return current->data;
18    }
19    bool operator==(const slist_iterator& it) const { return current == it.current; }
20    bool operator!=(const slist_iterator& it) const { return current != it.current; }
21};
22
23 template<typename T> class slist {
24     Node<T>* head=nullptr;
25     public:
26        using iterator = slist_iterator<T>;
27
28        void push_front(const T& data) {
29            head = new Node<T>(data, head);
30        }
31
32        iterator begin() { return iterator{head}; }
33        iterator end() { return iterator{nullptr}; }
34};
35
36 template<typename T1, typename T2>
37 constexpr T1 find(T1 first, T1 last, const T2& value);
38 for(; first != last; ++first) {
39     if(*first == value) return first;
40 }
41 return last;
42 }
43
44 int main() {
45     slist<int> s;
46     s.push_front(10);
47     s.push_front(20);
48     s.push_front(30);
49     s.push_front(40);
50     s.push_front(50);
51
52     auto ret = find(s.begin(), s.end(), 30); // 반복자 덕분에 find() 알고리즘이 정상적으로 동작한다
53
54     if(ret == s.end()) std::println("fail");
55     else             std::println("{}", *ret);
56 }

```

- 배열 뿐만 아니라 모든 컨테이너에서 선형 검색을 수행할 수 있다. ([진정한 Generic Algorithm이 되었다](#))

3.1.4 Container, Iterator, Algorithm (CIA)

컨테이너 안에 1이 몇 개 있는지 알고 싶다.

```

1 int main() {
2     std::vector c{1,2,3,1,2,3,1,2,3,1};
3     std::list   c{1,2,3,1,2,3,1,2,3,1};
4
5     int n = std::count(c.begin(), c.end(), 1); // std::count 알고리즘을 사용하면 된다. 반복자만 전달해주면
6         // vector, list 상관없이 사용 가능하다.
7     std::println("{}",n);

```

8 }

알고리즘의 인자로 "반복자"가 아닌 "컨테이너"를 보낼 수는 없을까?

```
1 int main() {
2     std::vector c{1,2,3,1,2,3,1,2,3,1};
3
4     int n1 = std::count(c.begin(), c.end(), 1); // c++98 스타일
5
6     int n2 = std::ranges::count(c, 1);           // c++20 스타일
7     int n3 = std::ranges::count(c.begin(), c.end(), 1);
8
9     using namespace std::ranges;                 // namespace 생략
10    int n4 = count(c, 1);
11    int n5 = count(c.begin(), c.end(), 1);
12
13    std::println("{}", n);
14 }
```

- c++98 시절에는 기술적인 문제로 컨테이너를 인자로 받는 알고리즘을 만들 수 없었다.

- c++20에 concept이 추가되면서 std::ranges에 컨테이너를 인자로 받는 알고리즘들이 대부분 구현되어 있다.

3.1.5 Member type

T가 컨테이너일 때 요소의 탑입을 알고 싶다.

```
1 template<typename T, typename Ax = std::allocator<T>>
2 class list {
3 public:
4     using value_type = T; // 컨테이너 규칙 상 value_type으로 탑입을 외부에 알려줘야 한다!
5 ...
6 }
7
8 list<int> s = {1,2,3,4,5};
9 list<int>::value_type n = s.front();
```

다음 예제와 같이 value_type은 함수 안에서 유용하게 사용할 수 있다.

```
1 template<typename T>
2 void print_first_element(const T& v) {
3     typename T::value_type n = v.front(); // typename을 앞에 추가해줘서 값이 아니라 탑입임을 명시한다!
4     std::println("{}", n);
5 }
6
7 int main() {
8     std::list<int> c{1,2,3,4,5};
9     print_first_element(c);
10 }
```

T::value_type 앞에 typename은 왜 붙이는 걸까? 다음 코드를 보자.

```
1 struct Test {
2     static constexpr int value = 10;
3     using DWORD = int;
4 };
5 int p1 = 0;
6
7 template<typename T>
8 void foo(T a) {
9     // 아래 코드에서 * 연산자의 의미는?
10     T::value* p1; // 곱셈
11     T::DWORD* p2; // 포인터 변수 선언
12 }
```

- T:: 다음에는 값 또는 타입이 올 수 있다. 값이 오면 *는 곱셈이 되고 타입이 오면 *는 포인터 변수의 선언이 된다!

dependent name

- 템플릿 인자 T에 의존해서 꺼내는 이름 (T::Name)
- T는 임의의 타입이므로 컴파일러가 Name의 의미를 조사할 수 없다
- 따라서 타입은 앞에 `typename T::` 을 사용하여 타입이라고 명시해줘야 한다!

```
1 struct Test {
2     static constexpr int value = 10;
3     using DWORD = int;
4 };
5 int p1 = 0;
6
7 template<typename T>
8 void foo(T a) {
9     T::value* p1;
10    typename T::DWORD* p2; // typename 선언
11 }
```

3.2 Iterator

3.2.1 Iterator concept

반복자(iterator)란 컨테이너의 "내부구조에 상관없이 동일한 방법으로 모든 요소에 순차적으로 접근"하기 위한 객체이다. 반복자를 사용하면 컨테이너의 내부구조를 몰라도 컨테이너의 모든 요소에 동일한 방법으로 접근할 수 있다.

STL의 반복자는 포인터와 동일한 방법으로 사용할 수 있도록 설계되어 있다. 포인터와 동일하게 `++p`, `p++`, `*p`를 사용하여 증가, 감소, 요소에 접근할 수 있다.

```
1 int main() {
2     std::list s{1,2,3,4};
3     std::vector v{1,2,3,4}; // list, vector는 서로 다른 자료구조이지만
4
5     auto si = s.begin();    // 동일한 방법으로 반복자를 얻어서
6     auto vi = v.begin();
7
8     ++si, ++vi;           // 동일한 방법으로 이동하고 요소에 접근할 수 있다.
9     std::println("{} {}", *si, *vi);
10 }
```

- 객체가 iterator 요구 조건을 만족하는지 확인하는 방법
- c++20에서 추가된 concept 라이브러리를 사용하면 된다.
- `#include <iterator>`

Concept	요구 조건
indirectly_readable	<code>==it</code>
indirectly_writable	<code>*it =</code>
weakly_incrementable	<code>++it && it++</code>
incrementable	<code>weakly_incrementable && regular</code>
input_iterator	<code>weakly_incrementable && indirectly_readable</code>
output_iterator	<code>weakly_incrementable && indirectly_writable</code>
input_or_output_iterator	<code>input_iterator output_iterator</code>

다음은 concept을 사용하여 반복자의 타입을 판별하는 코드이다.

```
1 template<typename T>
2 void check(const T& it) {
3     std::println("{}", std::input_or_output_iterator<T> );
```

```

4 }
5
6 int main() {
7     int x[4]{1,2,3,4};
8     std::vector v{1,2,3,4};
9
10    int* p1 = x;
11    auto p2 = v.begin();
12
13    check(p1); // true
14    check(p2); // true
15
16    int n=0;
17    check(n); // false
18 }
```

3.2.2 Container iterator

컨테이너의 반복자 타입은 컨테이너 이름<타입>::iterator와 같이 선언한다.

```

1 std::vector<int>::iterator p = c.begin(); // 코드 변경 시 해당 줄도 변경해야 하므로 auto 사용이 권장된다
2 auto p = c.begin();
```

컨테이너 반복자의 end()는 마지막이 아닌 "마지막 다음 요소"를 가리킨다. (Past-the-last element)

- * 연산자로 요소에 접근하면 안된다.
- 반복문 등에서 끝에 도달했는지 확인하는 용도로 사용된다.

컨테이너에서 반복자를 꺼내는 3가지 방법은 다음과 같다

```

1 auto p1 = c.begin();           // c++98, c가 배열이라면 사용 불가능
2 auto p2 = std::begin(c);      // c++11, c가 배열이라도 사용 가능
3 auto p3 = std::ranges::begin(c); // c++20, 기존 방법보다 안전한 방법
```

```

1 void f1(auto& c) { // c가 배열이라면 사용 불가능
2     auto p = c.begin();
3     auto s = c.size();
4 }
5
6 void f2(auto& c) {
7     auto p = std::begin(c);
8     auto s = std::size(c);
9 }
10
11 int main() {
12     std::vector c{1,2,3,4};
13     int c[] {1,2,3,4};
14
15     f1(c); // c가 배열이라면 사용 불가능!
16     f2(c); // 모든 c에 대해 사용 가능
17 }
```

STL의 `std::begin()`는 컨테이너와 배열 버전이 각각 구현되어 있다.

```

1 template<typename C>
2 constexpr auto mybegin(C& c) noexcept(noexcept(c.begin()) -> decltype(c.begin())) { // c가 컨테이너인
3     경우
4     return c.begin();
5 }
6
6 template<typename T, std::size_t SZ> // c가 배열인 경우
7 constexpr T* mybegin(T(&arr)[SZ]) noexcept {
8     return arr;
9 }
```

```

10
11 int main() {
12 int x[]{1,2,3,4};
13 std::vector c{1,2,3,4};
14
15 auto p1 = mybegin(c);
16 auto p2 = mybegin(x);
17 }
```

3.2.3 Iterator invalidation

반복자를 할당받은 후 컨테이너의 크기를 변경하면(resize) 어떻게 되는지 보자.

```

1 int main() {
2 int n = 0;
3 std::vector c{1,2,3,4};
4
5 auto it = c.begin();
6 n = *it;
7
8 c.resize(6); // 내부 버퍼 메모리 재할당
9 n = *it;      // error! invalid!
10 }
```

- 위 현상을 반복자 무효화(invalidation)라고 한다
- 컨테이너에서 반복자를 얻은 후 다양한 멤버 함수의 호출 결과로 인해 **내부 구조(메모리)가 변경된 경우 이전에 꺼내 놓은 반복자는 더 이상 사용할 수 없다.**
- 반복자는 언제 무효화 되는가? 컨테이너의 종류, 멤버 함수의 종류에 따라 다르다.
- 위 예시에서 vector의 크기를 줄이면 내부 구조가 변하지 않기 때문에 무효화되지 않는다. 크기를 키우는 경우 메모리를 재할당하면서 무효화된다!
- cppreference.com에 들어가 보면 컨테이너 별로 어떤 함수를 불렀을 때 무효화되는지 자세하게 나와있다

반복자의 크기와 함수 인자를 call by value로 작성하는 것

- 반복자는 크기가 크지 않고 복사 생성자 호출에 따른 오버헤드도 거의 없기 때문에 call by value로 함수 인자를 사용하는 것이 코딩 관례이다.

```

1 template<typename ITER, typename T>
2 ITER myfind(ITER first, ITER last, const T& value) { // 반복자는 call by value로 받는다
3     return first;
4 }
5
6 int main() {
7     std::println("{}", sizeof(std::list<int>::iterator)); // 8 byte
8     std::println("{}", sizeof(std::vector<int>::iterator)); // 8 byte
9     std::println("{}", sizeof(std::deque<int>::iterator)); // 24 byte
10
11    std::vector v{1,2,3,4};
12    auto ret = myfind(v.begin(), v.end(), 3);
13 }
```

컨테이너의 모든 요소를 화면에 출력하려면 range-for 구문을 사용하거나(가장 간결하고 좋은 방식) 반복자를 사용한다.

```

1 int main(){
2     std::vector v{1,2,3,4,5};
3
4     for(const auto& e : v) { // 1
5         std::print("{} ", e);
6     }
7
8     auto first = v.begin(); // 2
9     auto last = v.end();
10    while(first != last) {
```

```
11     std::print("{} , ", *first++);
12 }
13 std::println("");
14 }
```

3.2.4 std::ranges::begin

`std::ranges::begin()`은 C++20에서 새로 추가되었다. `std::begin()`과 `std::ranges::begin()`의 차이점에 대해 알아보자.

```
1 #include <vector>
2 #include <ranges> // std::ranges 네임스페이스 코드들이 구현되어 있음
3
4 int main(){
5     std::vector v{1,2,3,4,5};
6
7     auto it1 = std::begin(v);
8     auto it2 = std::ranges::begin(v);
9 }
```

`std::begin()`, `std::end()` 컨테이너 버전의 원리는 "컨테이너의 멤버 함수를 다시 호출"하는 것이다.

- 반환 타입이 "반복자(iterator)"가 아니라도 문제 없다.
- `std::ranges::begin()`, `std::ranges::end()`는 **반환 타입이 반복자인지 검사한다**.

```
1 class MyType {
2 public:
3     int begin() { return 0; }
4     int end() { return 0; }
5
6     int* begin() { return 0; }
7     int* end() { return 0; }
8 }
9
10 int main(){
11     MyType c;
12
13     auto it1 = std::begin(c);          // int, int* 타입 모두 작동한다
14     auto it2 = std::ranges::begin(c); // int 반환 타입인 경우 에러 발생한다
15 }
```

또 다른 예제를 보자.

```
1 int main(){
2     auto it1 = std::begin( std::vector{1,2,3} );           // 컨테이너에 임시 객체를 넘기는 경우
3
4     *it1 = 10;                                         // error. it1은 const iterator
5     int n = *it1;                                       // ok. 하지만 임시 객체는 파괴되었으므로 잘못된 주소를
6     // 가리키고 있다(dangling).
7
8     auto it2 = std::ranges::begin( std::vector{1,2,3} ); // rvalue인지 조사하는 과정에서 컴파일 에러가
9     // 발생한다
10 }
```

- `std::ranges::begin(c)`은 컴파일 과정에서 `c`가 borrowed ranges가 아니고 rvalue라면 컴파일 에러를 반환한다!

borrowed ranges

```
1 int main(){
2     std::string s{"to be or not to be"};
3
4     std::string ss = s; // 깊은 복사
5     std::string_view sv = s; // 복사하지 않고 s를 가리킴
6 }
```

```
7 auto it1 = std::ranges::begin( std::vector{1,2,3} ); // error
8 auto it2 = std::ranges::begin( std::string_view{s} ); // ok. it2는 s를 가리킨다
9 }
```

- `std::string`은 자원(문자열)을 소유한다.
- `std::string_view`는 자원을 소유하지 않고 다른 객체의 자원을 빌려서 사용한다 (대표적인 borrowed ranges)

c++20에서 추가된 용어와 개념

- Container: c++98부터 사용되던 용어
- View: c++20에서 추가된 용어. `std::string_view`의 개념을 vector, list 등으로 확장한 개념
- Range: c++20에서 추가된 용어. begin, end로 반복자를 꺼낼 수 있는 모든 타입을 말한다

3.2.5 Iterator value type

컨테이너의 구간 합을 구하는 코드를 작성해보자.

```
1 template<typename T>
2 auto sum(T first, T last) {
3     ? s = 0;                                // 이 곳의 타입을 어떻게 선언할 것인가?
4     while(first != last) { s += *first++; }
5     return s;
6 }
7
8 int main() {
9     std::vector v{1,2,3,4,5,6,7,8,9,10};
10
11    auto s = sum(v.begin(), v.end());
12
13    std::println("{}", s);
14 }
```

- T가 반복자일 때 T가 가르키는 요소의 타입은 어떻게 알 수 있을까?

- c++20: `std::iter_value_t<T>`
- c++98: `typename std::iterator_traits<T>::value_type`

반복자를 만들 때 `using value_type = T;`를 사용하여 value type을 명시해줘야 한다.

```
1 template<typename T> class vector_iterator {
2 public:
3     using value_type = T;
4 ...
5 };
```

```
1 template<typename T>
2 auto sum(T first, T last) {
3     typename T::value_type s = 0;
4     while(first != last) { s += *first++; }
5     return s;
6 }
7
8 int main() {
9     std::vector v{1,2,3,4,5};
10    int v[] {1,2,3,4,5};
11
12    auto s = sum(std::begin(v), std::end(v)); // v가 int*인 경우 typename T::value_type에서 에러가 발생한다!
13    std::println("{}", s);
14 }
```

- `typename T::value_type`는 에러가 발생하므로 `typename std::iterator_traits<T>::value_type`를 사용하여 컨테이너의 타입을 추론하였다.

- c++20에서 `std::iter_value_t<T>`는 결국 `using iter_value_t = typename std::iterator_traits<T>::value_type`

으로 축약한 것이다!

```
1 template<typename T> struct iterator_traits { // T가 컨테이너인 경우
2     using value_type = typename T::value_type;
3     ...
4 };
5
6 template<typename T> struct iterator_traits<T*> { // T가 포인터일 경우도 고려되어 설계되어 있다.
7     using value_type = T;
8     ...
9 };
```

3.2.6 Iterator category

아래와 같은 더블 링크드 리스트와 싱글 링크드 리스트 코드를 보자.

```
1 int main() {
2     std::list      ds{1,2,3,4}; // double linked list
3     std::forward_list ss{1,2,3,4}; // single linked list
4
5     auto p1 = ds.begin();
6     auto p2 = ss.begin();
7
8     ++p1; // ok
9     ++p2; // ok
10
11    --p1; // ok
12    --p2; // error. 자료구조 특징 상 존재할 수 없는 연산자
13 }
```

STL에서는 반복자가 할 수 있는 능력에 따라 6가지로 분류한다.

Category	Requirement	Container
output_iterator	write	
input_iterator	read, ++ (w/o multiple pass)	
forward_iterator	read, ++ (w/ multiple pass)	forward_list
bidirectional_iterator	read, ++, --	list
random_access_iterator	read, ++, --, +, -, []	
contiguous_iterator	read, ++, --, +, -, [] contiguous storage	vector
input_or_output_iterator	input_iterator output_iterator	

- 대표적인 질문이 "forward_iterator로는 write는 할 수 없나요?"인데 이는 안되는 것이 아니라 "[요구 조건이 아니라는 의미](#)"이다.

```
1 int main() {
2     std::list      ds{1,2,3,4};
3     std::forward_list ss{1,2,3,4};
4     std::vector    ve{1,2,3,4};
5
6     std::reverse( ds.begin(), ds.end() );
7     std::reverse( ss.begin(), ss.end() ); // error! reverse 연산자는 반드시 - 연산이 가능해야 한다
8     std::reverse( ve.begin(), ve.end() );
9 }
```

Iterator category가 왜 중요한가?

- STL의 다양한 알고리즘은 "[인자로 전달되는 반복자에 대한 요구 조건](#)"을 가지고 있다. 요구 조건은 카테고리로 제시되기 때문에 어떤 컨테이너가 어떤 카테고리에 속하는지 알아야 한다.
- STL 카테고리 관련 컴파일 에러는 매우 복잡하기 때문에 미리 이에 대한 지식을 알고 있으면 훨씬 수월하게 코딩할 수 있다.

Multiple passes란?

- `it1`, `it2`가 컨테이너의 같은 요소를 가르키는 반복자일 때 `*it1 == *it2`를 만족하고 `++it1`, `++it2` 후에 다시 `*it1 == *it2`를 만족하는 것을 말한다
- "주어진 구간을 2개 이상의 반복자가 지나가도 동일한 값을 읽을 수 있는 것". 하나의 반복자로 읽기를 수행해도 요소의 값이 변하지 않는 것을 말한다
- `std::istream_iterator`는 입력 버퍼의 요소를 읽을 때 값을 꺼내기 때문에 multiple passes가 아니다.

특정 반복자가 category를 만족하는지 조사하려면 c++20의 Concept을 사용하면 된다.

```
1 template<typename T> void p() {
2     std::println("input: {0}, forward: {1}",
3     std::input_iterator<T>,
4     std::forward_iterator<T>);
5 }
6
7 int main() {
8     p<std::vector<int>::iterator>();
9     p<std::list<int>::iterator>();
10    p<std::istream_iterator<int>::iterator>();
11 }
```

`std::find()`와 `std::remove()` 함수를 보자.

```
1 int main() {
2     std::vector v{1,2,3,3,4,3};
3
4     auto it1 = std::find(v.begin(), v.end(), 4);
5     auto it2 = std::remove(v.begin(), v.end(), 3);
6 }
```

- `std::find`는 value를 찾을 때까지 `++연산`으로 이동만 하기 때문에 반복자가 multiple pass를 보장할 필요가 없다.
(입력 반복자가 input iterator이기만 하면 됨)

- `std::remove`는 실제 컨테이너의 항목을 제거하는 것이 아니라 value를 찾으면 뒤의 요소를 앞으로 복사(이동)
하는 연산을 수행한다.

- 따라서 `++연산`이 가능해야 하며 multiple pass를 보장해야 한다.

앞서 사용했던 `p()` 함수를 좀 더 발전시키면 반복자의 category를 정확하게 알 수 있다.

```
1 template<typename T>
2 void p(std::string_view name) {
3     if constexpr( std::contiguous_iterator<T> )
4         std::println("{:6} : contiguous_iterator", name);
5
6     else if constexpr( std::random_access_iterator<T> )
7         std::println("{:6} : random_access_iterator", name);
8
9     else if constexpr( std::bidirectional_iterator<T> )
10        std::println("{:6} : bidirectional_iterator", name);
11
12    else if constexpr( std::forward_iterator<T> )
13        std::println("{:6} : forward_iterator", name);
14
15    else if constexpr( std::input_iterator<T> )
16        std::println("{:6} : input_iterator", name);
17 }
18
19 int main() {
20     p<std::vector<int>::iterator>("vector");
21     p<std::list<int>::iterator>("list");
22     p<std::deque<int>::iterator>("deque");
23     p<std::array<int,5>::iterator>("array");
24     p<std::string::iterator>("string");
```

```

25 p<std::set<int>::iterator>("set");
26 p<std::map<int, int>::iterator>("map");
27 p<std::forward_list<int>::iterator>("forward_list");
28 }

```

3.2.7 Iterator operation

다음과 같이 벡터의 4번째 요소에 접근하는 코드를 작성해보자

```

1 void foo(auto it) {
2     it = it+3;
3     std::println("{}", *it); // 4 출력
4 }
5
6 int main() {
7     std::vector v{1,2,3,4,5,6,7,8,9,10}; // ok
8     std::list v{1,2,3,4,5,6,7,8,9,10}; // error!
9     foo(v.begin());
10 }

```

만약 vector가 아닌 list였다면 위 코드가 동작할까? list는 임의 접근반복자(random access iterator)가 아니기 때문에 `it = it+3` 코드가 동작하지 않는다.

- `++it, ++it, ++it`와 같이 입력할 수 있으나 보다시피 코드가 비효율적이다.
- `foo`를 범용적인 함수로 만들기 위해 어떤 반복자가 들어와도 N 만큼 전진하는 코드를 작성해보자
- `std::advance(iterator, N)` 을 사용하면 iterator를 N번 전진한다. (vector, list 모두 동작!)

```

1 void foo(auto it) {
2     std::advance(it, 3); // vector, list 등 모든 컨테이너에 동작한다
3     std::println("{}", *it); // 4 출력
4 }
5
6 int main() {
7     std::vector v{1,2,3,4,5,6,7,8,9,10}; // ok
8     std::list v{1,2,3,4,5,6,7,8,9,10}; // ok
9     foo(v.begin());
10 }

```

`std::advance`와 유사한 몇가지 함수에 대해 알아보자 (`std::prev, std::next, std::distance`)

```

1 int main() {
2     std::list s{1,2,3,4,5,6,7,8,9,10};
3
4     auto it1 = s.begin();
5
6     std::advance(it1, 3); // std::advance
7     std::println("{}", *it1);
8
9     auto it2 = std::prev(it1, 3); // std::prev
10    auto it3 = std::next(it1, 3); // std::next
11    std::println("{} {}, {}", *it1, *it2, *it3);
12
13    auto n = std::distance(it2, it3);
14    std::println("{} {}", n); // std::distance
15 }

```

- `std::prev, std::nexts`는 반복자 자체는 변하지 않고 이동한 반복자를 반환한다
- `std::distance`는 두 반복자 사이의 거리를 반환한다

지금까지 배운 알고리즘들을 c++ 버전 별로 정리해보자(c++98, c++11, c++20)

```

1 int main() {
2     std::list s{1,2,3,4,5,6,7,8,9,10};

```

```

3
4 auto first = s.begin();
5
6 // std::next
7 auto it1 = std::next(first);           // 한 칸 전진
8 auto it2 = std::next(first, 3);
9
10 // std::ranges::next
11 auto it3 = std::ranges::next(first);
12 auto it4 = std::ranges::next(first, 3);
13 auto it5 = std::ranges::next(first, s.end()); // 무조건 s.end()를 반환 (추후 counted iterator를 쓸 때
     유용하게 사용)
14 auto it6 = std::ranges::next(first, 3, s.end()); // 3칸 이동하거나 끝 중 먼저 도달하는 것을 반환
15 auto it7 = std::ranges::next(first, 20, s.end()); // 20칸 이동하거나 끝 중 먼저 도달하는 것을 반환
16
17 std::println("{}", it6 == it4); // true
18 std::println("{}", it5 == it7); // true
19 }
```

- c++98: `std::advance`, `std::distance`
- c++11: `std::prev`, `std::next`
- c++20: `std::ranges::advance`, `std::ranges::next`, `std::ranges::prev`, `std::ranges::distance`

3.2.8 ++ vs next

컨테이너에서 2번째 요소를 가리키는 반복자가 필요할 때 `++`, `next` 중 어떤 것이 더 좋은 방법일까?

```

1 int main() {
2     std::vector c{1,2,3,4};
3
4     auto it1 = c.begin() + 1;           // #1
5     auto it2 = ++c.begin();          // #2
6     auto it3 = std::next(c.begin()); // #3
7     auto it4 = std::ranges::next(c.begin()); // #4
8 }
```

- `c.begin() + 1`은 반복자가 임의의 접근 반복자인 경우만 가능하다. (`list`인 경우 에러 발생)
- `auto it = ++c.begin()`은 대부분 성공하지만 컴파일 에러가 발생할 수도 있다.
- 따라서 `std::next`, `std::ranges::next`를 사용하는 것을 권장한다!

`++c.begin()`이 에러가 발생하는 경우에는 어떤 것들이 있을까?

```

1 int next(int addr) { return ++addr; }
2 int x[5]{1,2,3,4,5};
3 int* address_of_x() { return &x[0]; }
4
5 int main() {
6     int* p1 = ++( &x[0] );           // error
7     int* p2 = ++( address_of_x() ); // error
8
9     int* addr = &x[0];
10    int* p3 = ++addr;              // ok. addr에 담기면서 lvalue가 되므로 컴파일된다
11    int* p4 = next( &x[0] );       // ok
12
13    std::println("{} , {}", *p3, *p4);
14 }
```

- `++(&variable)`: 연산자가 반환하는 주소 값은 "rvalue"이다. rvalue는 `++` 연산을 할 수 없다 (compile error!)

임의의 유저 타입에 대한 반복자 코드를 살펴보자.

```

1 class myiterator {
2     int* current;
3 public:
```

```

4 myiterator(int* p = nullptr) : current{p} {}
5
6 myiterator& operator++() { ++current; return *this; }
7 int& operator*() { return *current; }
8 };
9
10 int x[5]{1,2,3,4,5};
11
12 int* get_raw_pointer() { return &x[0]; }
13 myiterator get_myiterator() { return &x[0]; }
14
15 int main() {
16     auto it1 = ++get_raw_pointer(); // error!
17     auto it2 = ++get_myiterator(); // ok
18     std::println("{}", *it2);
19 }
```

- 왜 굳이 raw pointer와 동일한 동작을 하는(++, *) 반복자 클래스를 작성했을까?
- ++(get_raw_pointer())의 반환값은 "주소 값, rvalue"이다. 따라서 ++ 연산을 할 수 없다. (**compile error!**)
- ++(get_myiterator())의 반환값 또한 "myiterator 타입의 임시 객체, rvalue"이다. **하지만 rvalue는 등호 왼쪽에 놓일 수는 있지만 멤버 함수를 호출할 수 있다.** 즉, ++연산은 내부적으로 () .operator++() 함수를 호출하게 된다.

```

1 class myiterator {
2     int* current;
3 public:
4     myiterator(int* p = nullptr) : current{p} {}
5
6     myiterator& operator++() { ++current; return *this; }
7     int& operator*() { return *current; }
8 };
9
10 class MyContainer {
11     int data[5]{1,2,3,4,5};
12 public:
13     using iterator = myiterator; // 정상 동작한다. 만약 int*로 되어 있었다면 ++.mc.begin()에서 에러가 발생한다.
14
15     iterator begin() { return data; }
16     iterator end() { return data+5; }
17 };
18
19 int main() {
20     MyContainer mc;
21     auto it = ++mc.begin(); // ok
22     std::println("{}", *it);
23 }
```

3.2.9 std::copy vs std::ranges::copy

컨테이너 c1의 모든 요소를 c2에 복사하고 싶다고 가정하자.

```

1 int main() {
2     std::vector c1{1,2,3,4,5};
3     std::vector c2{0,0,0,0,0,0,0};
4
5     c2 = c1; // #1
6
7     c2.assign(c1.begin(), c1.end()); // #2
8
9     for( int i=0; i<c1.size(); i++ ) // #3
10     c2[i] = c1[i];
11
12     std::copy(c1.begin(), c1.end(), c2.begin());
13 }
```

- **대입연산자 사용** (`c2 = c1`): `c1`, `c2`가 동일 컨테이너여야 한다. `c2`의 기존 요소가 완전히 제거되고 `c1`을 복사한다.
- **assign 멤버 함수 사용**: `c1`, `c2`가 다른 컨테이너여도 사용 가능하다. `c2`의 기존 요소가 완전히 제거되고 `c1`을 복사한다.
- **반복문과 [] 연산자**: `c1`, `c2`가 `std::list`라면 [] 연산자 지원이 안된다. `c2`에서 `c1`의 크기만큼만 요소가 복사된다.
- **`std::copy`, `std::ranges::copy`**: 위 방법과 동일한 복사가 되고 `std::list` 포함한 대부분의 컨테이너에서 실행 가능하다.

`std::copy` - 128) * 64 + (' - 128) 뺀 `std::ranges::copy` 알고리즘은 무엇이 다를까? `std::copy (c++98)`: 인자로 반복자만 전달이 가능하다. 반환값은 출력 컨테이너의 반복자이다. `std::ranges::copy (c++20)`: 반복자 버전과 컨테이너 버전을 제공한다. 반환 값은 `in_out_result`이다.

```

1 int main() {
2     std::vector c1{1,2,3,4,5};
3     std::vector c2{0,0,0,0,0,0,0,0};
4
5     std::copy(c1.begin(), c1.end(), c2.begin());           // 반복자만 전달 가능
6
7     std::ranges::copy(c1.begin(), c1.end(), c2.begin()); // 반복자 뿐만 아니라
8     std::ranges::copy(c1, c2.begin());                   // 컨테이너도 전달 가능
9 }
```

두 알고리즘은 반환값도 서로 다르다.

```

1 int main() {
2     std::vector c1{1,3,5,7,9};
3     std::vector c2{2,4,6,8,10};
4
5     auto ret1 = std::copy(c1.begin(), std::next(c1.begin(),3), c2.begin()); // c2.end() 값을 반환한다
6     std::println("{}", *ret1);
7
8     auto ret2 = std::ranges::copy(c1.begin(), std::next(c1.begin(),3), c2.begin()); // (in and out)
9         ret2.in은 c1.end()를 반환하고 ret2.out는 c2.end()를 반환한다
10    std::println("{}", {}, *(ret2.in), *(ret2.out));
11 }
```

3.2.10 Reverse iterator

반대 방향으로 이동하는 반복자인 reverse iterator에 대해 알아보자

```

1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     auto it = std::next(v.begin(), 5); // 6
5
6     std::reverse_iterator ri{ it };
7
8     std::println("{} , {}", *it, *ri); // 6, 5
9     it++;
10    ri++;
11    std::println("{} , {}", *it, *ri); // 7, 4
12 }
```

- `ri++`를 수행하면 한 칸 뒤로 이동한다

reverse iterator는 기존의 방향을 반대로 해주는 기능을 가진 "iterator adaptor"이다.

- `++ri == --it`
- `--ri == ++it`
- `*ri = *std::prev(it)`: 현재 가리키는 곳에서 한 칸 뒤에 값을 반환한다.
- 양방향 반복자(bidirectional iterator) 조건을 만족하는 반복자에만 사용 가능하다.

reverse iterator 객체를 생성하는 방법은 두 가지가 있다. (`#include <iterator>` 해더 필요)

```

1 #include <iterator>
2
3 int main() {
4     std::vector v{1,2,3,4,5,6,7,8,9,10};
5
6     auto it = std::next(v.begin(), 5);
7
8     std::reverse_iterator ri{ it };           // #1
9     std::reverse_iterator<std::vector<int>::iterator> ri2{ it }; // c++17 이전에는 타입을 명시해줘야 한다.
10
11    auto ri3 = std::make_reverse_iterator(it);      // #2
12 }
```

또한 reverse iterator에 접근하기 위해서는 `rbegin`, `rend`를 사용해야 한다.

```

1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     auto first = v.begin();
5     auto last = v.end();
6
7     auto rfirst = v.rbegin(); // std::make_reverse_iterator(v.end())와 동일
8     auto rend = v.rend();   // std::make_reverse_iterator(v.begin())와 동일
9
10    std::vector<int>::reverse_iterator ri = v.begin();
11 }
```

- reverse iterator는 컨테이너에서 `{name}::reverse_iterator`로 바로 사용할 수 있다.

`const iterator`는 값을 변경할 수 없는 반복자를 말한다. (상수 버전 포인터와 동일). reverse iterator 또한 const 버전이 존재한다.

```

1 int main() {
2     std::vector v{1,2,3,4,5};
3
4     std::vector<int>::iterator it1 = v.begin();
5     std::vector<int>::reverse_iterator rit1 = v.rbegin();
6     std::vector<int>::const_iterator it1 = v.cbegin();
7     std::vector<int>::const_reverse_iterator rit1 = v.crbegin();
8
9     std::find(it1, rit1, 3); // error! 동일 타입이 아니므로 에러 발생
10 }
```

- const 타입 또한 멤버 함수가 아닌 일반 함수 버전과 `std::ranges` 버전도 모두 제공된다.

3.2.11 Insert iterator

선형 컨테이너 끝에 요소를 추가하는 2가지 방법이 있다

- 컨테이너의 `push_back()` 멤버 함수를 사용하는 방법
- 후방 삽입 반복자 (`std::back_insert_iterator`)를 사용하는 방법

```

1 int main() {
2     std::list s{1,2,3};
3
4     s.push_back(10);           // #1
5
6     std::back_insert_iterator bi{s}; // #2
7     *bi = 20;                 // s.push_back(20)과 동일
8     *bi = 30;                 // s.push_back(30)과 동일
9 }
```

-
- `std::back_insert_iterator`와 같은 반복자를 **삽입 반복자(insert iterator)**라고 한다 (c++98부터 지원)
 - 전방, 후방, 임의 삽입 등 3가지 형태로 제공된다.

삽입 반복자를 생성하는 방법에 대해 알아보자.

```
1 int main() {
2     std::list s{1,2,3};
3
4     std::back_insert_iterator bi1{s};           // c++17
5     std::back_insert_iterator<std::list<int>> bi2{s}; // c++17 이전에는 타입을 직접 명시해줘야 한다
6
7     auto bi3 = std::back_inserter(s);           // 편의 함수 템플릿(convienient function template)을
8         사용한다. (c++98부터 지원)
9
10    *bi1 = 10;
11    *bi2 = 20;
12    *bi3 = 30;
13 }
```

삽입 반복자와 copy 알고리즘에 대해 알아보자

- 컨테이너의 모든 요소를 다른 컨테이너 끝에 추가하고 싶다면
- 반복문과 `push_back()` 멤버 함수를 이용하는 방법과
- `copy` 알고리즘과 삽입 반복자를 이용하는 방법이 있다.

아래와 같이 벡터의 모든 요소를 리스트의 끝에 추가하고 싶은 경우를 보자.

```
1 int main() {
2     std::vector v{1,2,3,4,5};
3     std::list s1{0,0,0,0,0};
4     std::list s2{0,0,0,0,0};
5
6     for(auto e : v)                      // #1 push_back 사용
7         s1.push_back(e);
8
9     std::back_insert_iterator bi{s2};       // #2 copy 알고리즘과 후방 삽입 반복자 이용
10    std::ranges::copy(v, bi);
11
12    std::ranges::copy(v, std::back_inserter(s2));
13 }
```

(주의) `copy` 사용 시 컨테이너의 요소를 덮어 쓰는 것(overwrite)과 요소를 삽입(insert)하는 것을 잘 구별하고 사용해야 한다!

```
1 int main() {
2     std::vector v{1,2,3,4,5};
3     std::list s1{0,0,0,0,0};
4     std::list s2{0,0,0,0,0};
5
6     copy(v, s1.begin());                // 덮어쓰기(overwrite)
7     copy(v, std::back_inserter(s2)); // 삽입(insert)
8
9     std::list<int> s3;
10    copy(v, s3.begin());            // error! 요소가 없는 컨테이너에 복사하려고 한다.
11    copy(v, std::back_inserter(s3)); // ok! s3부터 값이 추가된
12 }
```

다음으로 삽입 반복자의 3가지 종류에 대해 알아보자.

- 임의 삽입 반복자는 생성자 인자로 2개를 전달해야 한다.
- `std::insert_iterator ii(container, iterator);`
- `auto ii = std::inserter(container, iterator);`
- iterator 앞쪽에 값을 넣겠다는 의미이다.

종류	형태
전방 삽입	<code>std::front_insert_iterator<></code> <code>std::front_inserter</code>
후방 삽입	<code>std::back_insert_iterator<></code> <code>std::back_inserter</code>
임의 삽입	<code>std::insert_iterator<></code> <code>std::inserter</code>

```

1 int main() {
2     std::list s{1,2,3,4,5};
3
4     // #1 클래스 템플릿 이용 (c++17 )
5     std::front_insert_iterator fi1{s};
6     std::back_insert_iterator bi1{s};
7     std::insert_iterator ii1{s, std::next(s.begin(), 2)}; // 3을 가리킴. 삽입을 수행하면 2와 3 사이에 삽입됨
8
9     // c++17 이전에는 타입을 명시해줘야 한다
10    std::front_insert_iterator<std::list<int>> fi2{s};
11    std::back_insert_iterator<std::list<int>> bi2{s};
12    std::insert_iterator<std::list<int>> ii2{s, std::next(s.begin(), 2)};
13
14    // #2 편의 함수 템플릿(convinent function template)을 사용한다
15    auto fi3 = std::front_inserter(s);
16    auto bi3 = std::back_inserter(s);
17    auto ii3 = std::inserter(s, std::next(s.begin(), 2));
18
19    *fi3 = 10;
20    *bi3 = 20;
21    *ii3 = 30;
22
23    show(s); // 10, 1, 2, 30, 3, 4, 5, 20 - 128) * 64 + (' - 128) 뿐
24        - 128) * 64 + (' - 128) 끝 - 128) * 64 + (' - 128) 끝.
}

```

전방 삽입 반복자와 임의 삽입 반복자의 차이점에 대해 알아보자

```

1 int main() {
2     std::vector v{1,2,3,4,5};
3     std::list s1{0,0,0,0,0};
4     std::list s2{0,0,0,0,0};
5
6     std::ranges::copy(v, std::front_inserter(s1));
7     std::ranges::copy(v, std::inserter(s2, s2.begin()));
8
9     show(s1); // 5,4,3,2,1,0,0,0,0,0
10    show(s2); // 1,2,3,4,5,0,0,0,0,0
11 }

```

- 전방 삽입 반복자를 사용하면 기존 값들이 **거꾸로** 들어간다. 하지만 임의 삽입 반복자는 **제대로** 들어간다.

다음으로 삽입 반복자를 사용할 때 주의 사항에 대해 살펴보자.

```

1 int main() {
2     std::vector v{1,2,3,4,5};
3
4     auto p1 = std::front_inserter(v); // error!
5     auto p2 = std::inserter(v, v.begin()); // ok
6     *p1 = 10;
7     *p2 = 10;
8
9     ++p2; // ok. ++연산자는 아무일도 하지 않는다
10
11 using T = std::back_insert_iterator<std::vector<int>>;

```

```
12 std::println("{}", std::output_iterator<T, int>);  
13 }
```

-
- `std::vector`는 전방 삽입 반복자를 사용할 수 있다.(`push_front`가 없음!)
 - 모든 반복자는 `++` 연산을 제공해야 한다. 단 삽입 반복자의 `++`은 아무 일도 하지 않는다.

삽입 반복자의 category는 **출력 반복자(output iterator)로 분류된다.**

3.2.12 `std::counted_iterator` & `std::default_sentinel`

counted iterator라는 개념은 c++20에서 추가되었으며 `<iterator>` 헤더에 포함되어 있다.

- counted iterator는 기존의 반복자에 추가적으로 "갯수를 관리하는 기능을 추가"한 반복자 어댑터이다.
- 동작 방식은 기존 반복자와 완전히 동일하다.

```
1 int main() {  
2     std::vector v{1,2,3,4,5,6,7,8,9,10};  
3  
4     auto it = v.begin();  
5  
6     std::counted_iterator ci{it, 5}; // it부터 5개까지만 접근할 수 있다는 의미  
7  
8     ++it;  
9     ++ci;  
10  
11    std::println("{} , {}" , *it, *ci); // 2, 2  
12  
13    std::println("{}" , ci.count()); // 4 (++ci에서 1 감소)  
14  
15    while(ci.count() != 0) {  
16        std::println("{}" , *ci++); // 2,3,4,5  
17    }  
18  
19    ++ci;  
20    std::println("{}" , ci.count()); // -1;  
21    std::println("{}" , *ci); // 이론적으로는 undefined이지만 실제 구현은 return *current가 된다. (사용  
22    안할 것을 권장)  
}
```

-
- `++ci` 연산 시 `++current`, `--count`가 내부적으로 행된다.
 - `count < 0` 인 경우 `*ci`를 하면 어떻게 될까? `count`는 계속 줄어들고 `current`는 `undefined`가 된다.

counted iterator를 다양한 알고리즘에 전달하려면 어떻게 해야 할까?

- 알고리즘에 반복자를 전달하려면 [first, last) 반복자 쌍이 필요하다. ex) `auto ret = std::find(ci, ???, 3);`
- counted iterator는 "한 개의 반복자 안에 범위의 끝에 대한 정보도 포함"하고 있다
- 하지만 std 기본 알고리즘은 반복자 쌍을 요구하므로 이 때 사용하는 것이 `default sentinel`이라는 객체이다.

```
1 int main() {  
2     std::vector v{1,2,3,4,5,6,7,8,9,10};  
3  
4     auto it = v.begin();  
5  
6     std::counted_iterator ci{it, 5};  
7  
8     // while(ci.count() != 0) // 이렇게 사용해도 되지만  
9     while(ci != std::default_sentinel) { // default sentinel을 써도 된다.  
10         std::println("{}" , *ci++);  
11     }  
12 }
```

default sentinel 구조체는 다음과 같이 정의되어 있다.

```

1 struct default_sentinel_t {};
2
3 inline constexpr default_sentinel_t default_sentinel{};
4
5 constexpr bool
6 operator==(const std::counted_iterator& ci,
7 std::default_sentinel_t) noexcept {
8 return ci.count == 0;
9 }

```

- == 연산자가 호출되면 ci.count가 0인지 검사한다.

c++98 시절에는 sentinel이라는 개념이 없었기 때문에 STL find 알고리즘이 다음과 같이 설계되어 있다.

```

1 template<class InputIt, class T>
2 InputIt find(InputIt first, InputIt last, const T& value);

```

- 구간을 나타내는 **반복자 쌍이 반드시 같은 타입**이어야 한다.

c++20에서 추가된 constrained 알고리즘 버전의 find는 다음과 같이 구현되어 있다.

```

1 template<std::input_iterator I,
2 std::sentinel_for<I> S,
3 class T, class Proj = std::identity >
4 requires
5 std::indirect_binary_predicate<ranges::equal_to,
6 std::projected<I, Proj>, const T*>
7
8 constexpr I find(I first, S last,
9 const T& value, Proj proj = {});

```

- 핵심은 구간을 나타내는 **반복자 쌍이 다른 타입도 가능하다**는 것이다.

```

1 int main() {
2 std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4 std::counted_iterator ci{v.begin(), 5};
5
6 auto ret = std::find(ci, std::default_sentinel, 3);      // error! 반복자 쌍이 반드시 같은 타입이어야 한다
7 auto ret = std::ranges::find(ci, std::default_sentinel, 3); // ok.
8
9 std::println("{}", *ret);
}

```

`std::counted_iterator`와 `std::default_sentinel`을 legacy function(c++98)에 전달하려면 어떻게 해야 할까?

- iterator와 sentinel의 "공통의 타입"을 만들어야 한다.
- `std::common_iterator`: iterator와 sentinel을 모두 담을 수 있는 공통의 타입을 설계할 때 사용한다.

```

1 int main() {
2 std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4 std::counted_iterator ci{v.begin(), 5};
5
6 using T = std::common_iterator<                                // 두 타입의 공통의 타입 선언
7 std::counted_iterator<std::vector<int>::iterator>,
8 std::default_sentinel_t>;
9
10 auto ret = std::find( T{ci}, T{std::default_sentinel}, 3);
11
12 std::println("{}", *ret);
}

```

3.2.13 Iterator and sentinel

find 알고리즘을 사용해서 선형 검색을 수행하려면 검색 대상 구간을 인자로 전달해야 한다.

c++98 <code>std::find</code>	시작과 끝(past the last element)를 나타내는 "2개의 반복자"를 전달한다. 반드시 같은 타입이어야 한다. <code>InputIt find(InputIt first, InputIt last, const T& value);</code>
c++20 <code>std::ranges::find</code>	시작을 나타내는 반복자와 끝을 나타내는 sentinel을 전달한다. sentinel은 반복자일 수도 있고 다른 형태의 객체일 수도 있다. <code>I find(I first, S last, ...);</code>

sentinel이라는 개념은 c++20에서 처음 제안된 개념으로 보통 반복자 구간의 끝을 나타낼 때 사용한다

- 사전적 의미1: 보초, 파수병
- 사전적 의미2: (특정 정보 블럭의 시작과 끝을 나타내는) 표지
- `std::default_sentinel_t`를 사용하려면 그 자체로는 "empty class"이므로 어떠한 정보도 담고 있지 않다
- **1번째 인자로 전달되는 iterator 자체에 구간 끝에 대한 정보가 있어야 한다.**

```

1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     auto it = v.begin();
5     std::counted_iterator ci{it, 5};
6
7     auto ret1 = std::ranges::find(ci, std::default_sentinel, 3); // ok
8     auto ret2 = std::ranges::find(it, std::default_sentinel, 3); // error! it에 구간 끝에 대한 정보가 없다
9     auto ret3 = std::ranges::find(it, v.end(), 3);           // ok
10 }
```

`std::sentinel_for<S, I>`: S가 I의 sentinel로 사용 가능한지를 조사할 때 사용하는 concept이다.

```

1 template<class S, class I>
2 concept sentinel_for =
3     std::semiregular<S> &&
4     std::input_or_output_iterator<I> &&
5     _WeaklyEqualityComparableWith<S, I>;
```

```

1 int main() {
2     using vi_t = std::vector<int>::iterator;
3     using ci_t = std::counted_iterator<vi_t>;
4
5     bool b1 = std::sentinel_for<std::default_sentinel_t, ci_t>; // true. default_sentinel_t를 벡터의 ci로
6     쓸 수 있다
7     bool b2 = std::sentinel_for<std::default_sentinel_t, vi_t>; // false. default_sentinel_t를 벡터의
8     반복자로 쓸 수 없다
9
10    std::println("{} {}", b1, b2);
11
12    bool b3 = std::sentinel_for<vi_t, vi_t>; // true
13 }
```

따라서 최신 STL find 알고리즘에는 sentinel을 고려하여 코드가 구현되어 있다.

```

1 template<typename InputIter>
2 void find_cpp98(InputIter first, InputIter last) {}
3
4 template<std::input_or_output_iterator I, std::sentinel_for<I> S>           // concept을 사용하여 구현되어
5     void find_cpp20(I first, S last) {}
```

3.2.14 Ostream iterator

반복자를 사용하여 화면에 숫자/문자열을 출력할 수 있다

```
1 int main() {
2     int n = 10;
3
4     std::cout << n << std::endl;           // #1
5
6     std::ostream_iterator<int> p(cout, ", "); // #2 반복자를 통한 출력
7     *p = 20;                                // cout << 20 << ", "과 동일
8     *p = 30;                                // cout << 30 << ", "과 동일
9
10    std::list<int> s = {1,2,3};
11    std::copy(std::begin(s), std::end(s), p); // 리스트에 있는 모든 내용을 p로 복사하는데 p가 출력 반복자이므로
12        // 화면에 출력됨
13    std::fill_n(p, 3, 0);                   // 출력 반복자에 0을 세 번 출력한다
14
15    ++p;                                  // ok. 아무 동작도 하지 않는다
}
```

- 스트림 반복자는 입출력 스트림에서 요소를 읽거나 쓰기 위한 반복자이다.
- **ostream_iterator**: 출력 스트림을 사용해서 출력을 하는 반복자로 copy 등의 알고리즘 함수를 사용해서 스트림에 출력할 때 사용한다
- delimiter(", ")를 추가해도 되고 없이 사용해도 된다.
- 삽입 반복자(insert iterator)와 마찬가지로 ++ 연산은 구현이 되어 있지만 아무 동작도 하지 않는다.

파일에 입출력을 하고 싶을 경우 다음과 같이 사용한다

```
1 int main() {
2     ofstream f("a.txt");
3
4     std::ostream_iterator<int> p(f, ", "); // 파일에 출력하는 반복자
5
6     *p = 20;
7     *p = 30;                            // 결과 값이 파일에 출력(작성)된다.
8 }
```

3.2.15 Ostreambuf iterator

ostream 반복자와 구분되는 ostreambuf iterator에 대해 알아보자

```
1 int main() {
2     std::ostreambuf_iterator<char> p(cout); // char 타입만 받고 delimiter 입력이 불가능하다 (wchar_t)
3     *p = 65;                                // 'A'가 출력된다
4 }
```

- ostreambuf 반복자는 <char>만 출력할 수 있으며 delimiter 입력이 불가능하다.
- ostreambuf에 대해 제대로 이해하려면 cout에 대한 깊은 이해가 먼저 필요하다

cout << 65 코드가 실행되면

- 출력 버퍼에 '6', '5' 문자를 넣는다 (ostream 클래스)
- 버퍼를 관리하는 streambuf 클래스에서 버퍼가 가득 차면 화면에 출력한다
- cout.rdbuf() 함수를 사용하면 streambuf의 포인터를 얻을 수 있다.

```
1 int main() {
2     std::cout << 65;
3
4     streambuf* buf = cout.rdbuf();
5     buf->sputc(65);                  // 아스키코드 A(65)가 출력된다
6 }
```

-
- ostream_iterator: stream(cout)을 사용해서 화면을 출력하는 반복자 (서식화된 출력)
 - ostreambuf_iterator: streambuf를 사용해서 화면을 출력하는 반복자 (문자 타입만 출력)
 - 문자열만 사용하는 경우 ostreambuf_iterator를 사용하는 것이 속도가 조금 더 빠르다
-

```
1 int main() {
2     std::ostream_iterator<int> p1(cout, " ");
3     *p1 = 10;
4
5     std::ostream_iterator<char> p2(cout.rdbuf()); // 아스키코드 출력
6     *p2 = 'A';
7
8     std::ostream_iterator<wchar_t> p3(wcout.rdbuf()); // 유니코드 출력
9     *p3 = L'A';
10 }
```

깊은 이해를 위해 ostream iterator의 구현 코드를 살펴보자.

```
1 template<typename T, typename CharT = char, typename Traits = char_traits<CharT>> class
2     eostream_iterator {
3     std::basic_ostream<CharT, Traits>* stream;
4     const CharT* delimiter;
5
6     public:
7     using iterator_category = output_iterator_tag;
8     using value_type = void;
9     using pointer = void;
10    using reference = void;
11    using difference_type = void;
12
13    using char_type = CharT;
14    using traits_type = Traits;
15    using ostream_type = std::basic_ostream<CharT, Traits>
16
17    eostream_iterator(ostream& os, const CharT* const deli=0) :stream(&os), delimiter(deli) {}
18
19    eostream_iterator& operator*() { return *this; }
20    eostream_iterator& operator++() { return *this; }
21    eostream_iterator& operator++(int) { return *this; }
22
23    eostream_iterator& operator=(const T& v) {
24        *stream << v;
25        if(deli != 0)
26            *stream << delimiter;
27        return *this;
28    };
29
30    int main() {
31        eostream_iterator<int> p(cout, " ");
32        *p = 10; // ( p.operator*() ).operator=(10) 호출
33    }
```

3.2.16 Istream iterator

출력 반복자와 반대로 값을 입력하는 istream iterator 또한 존재한다

```
1 int main() {
2     std::istream_iterator<int> p1(cin);
3     int n = *p1; // cin이 실행되어 값을 입력받는다
4
5     std::cout << n << std::endl;
6 }
```

ostreambuf iterator와 동일하게 istreambuf iterator 또한 존재한다. istream_iterator는 white space를 입력받지 못하지만 istreambuf_iterator는 white space를 입력받을 수 있다.

스트림 반복자	출력 대상	출력 형태
ostream_iterator	basic_ostream	서식화된 출력
ostreambuf_iterator	basic_ostreambuf	CharT 출력
istream_iterator	basic_istream	서식화된 입력
istreambuf_iterator	basic_istreambuf	CharT 입력

파일로부터 입력 반복자를 받아서 출력 반복자를 통해 화면에 출력하는 코드를 작성해보자

```
1 int main() {
2     std::ifstream f("a.txt");
3
4     std::istreambuf_iterator<char> p1(f), p2; // 디폴트 생성자 p2는 end of stream을 나타낸다.
5     std::ostream_iterator<char> p3(cout);
6
7     std::copy(p1, p2, p3); // p1부터 p2(end of stream)까지 p3 반복자에 복사한다. 즉 모든 파일
8     내용이 출력된다.
}
```

3.3 Algorithm

3.3.1 Erase-remove idioms

다음과 같이 벡터에서 3을 지우는 코드를 살펴보자

```
1 int main() {
2     std::vector v{1,2,3,1,2,3,1,2,3,1};
3
4     auto ret = std::remove(v.begin(), v.end(), 3);
5     show(v); // 1,2,1,2,1,2,3,1 이 나온다. 마지막 3이 지워지지 않았다.
6 }
```

- 마지막 원소 3은 왜 지워지지 않았을까?

std::remove는 다음과 같이 구현되어 있다.

```
1 template<typename ForwardIt, typename T>
2 ForwardIt remove(ForwardIt first,
3 ForwardIt last,
4 const T& value)
5 {
6     // [first, last) 구간에서 value를 제거
7 }
```

- remove 구현은 대부분의 컨테이너에서 동작할 수 있도록 코드가 작성되어 있다.
- remove 알고리즘은 내부적으로 반복자만 알 수 있고, 컨테이너는 알 수 없으므로 컨테이너의 크기 자체를 변경하지 않는다. (알고리즘은 컨테이너를 알지 못한다.)
- remove의 구현 원리는 조건을 만족하는 요소를 찾으면 뒤에 있는 요소를 현재 위치로 이동(move)한다. 즉 [1,2,1,2,1,2,1] 다음에 기존에 있던 [2,3,1]도 그대로 존재한다.

```
1 int main() {
2     std::vector v{1,2,3,1,2,3,1,2,3,1};
3
4     auto ret = std::remove(v.begin(), v.end(), 3);
5     show(v);
6
7     v.erase(ret, v.end()); // ret을 기준으로 뒷 부분을 제거하면
8     show(v); // 1,2,1,2,1이 제대로 출력된다.
9 }
```

- 한 줄로 줄여서 `v.erase(std::remove(v.begin(), v.end(), 3), v.end());` 로 쓰기도 한다. (**erase-remove 기술**)
- 연속된 메모리를 사용하는 vector에서는 가장 효율적인 방법이다.

`std::remove`과 `std::ranges::remove`의 차이점은 무엇일까?

```

1 template<typename ForwardIt, typename T>
2 ForwardIt remove(ForwardIt first,
3 ForwardIt last,
4 const T& value)
5 {
6     first = std::find(first, last, value);
7     if(first != last)
8         for(ForwardIt i = first, ++i != last; )
9             if(!(*i == value))
10                 *first++ = std::move(*i);    // std::move가 사용된다
11     return first;
12 }
```

- `std::remove()`는 `std::move()`를 사용하여 뒤에 있는 요소를 앞으로 이동(move)시킨다.

```

1 int main(){
2     std::vector v{1,3,1,3,1,3,10,3,9,3};
3
4     auto ret1 = std::remove(v1.begin(),
5                             std::next(v1.begin(), 7), 3); // 끝까지 지우는 것이 아닌 구간 내에서 3을 지우는 코드를 보자
6
7     show(v1); // 1,1,1,10,[1,3,10],3,9,30] 출력된다. 중간에 [1,3,10]이 잘
8     지워지지 않았다.
9
10    v1.erase(ret1, v1.end()); // 1,1,1,10만 출력된다. 하지만 우리가 원하는 것은 [1,3,10]을
11        제외한 값을 얻고 싶다
12    show(v1);
13
14    v1.erase(ret1, std::next(v1.begin(), 7)); // 1,1,1,10,3,9,30] 출력된다
15    show(v1);
16 }
```

- 결국 구간 내에서 제대로 remove-erase idiom을 사용하려면 `v1.erase(et1, std::next(v1.begin(), 7));`와 같이 제거해줘야 한다.
- `std::ranges::remove`를 사용하면 반환값이 `std::ranges::subrange`인데 여기에 v1의 잘라진 구간의 처음과 끝 정보가 포함되어 있다.

```

1 int main(){
2     std::vector v{1,3,1,3,1,3,10,3,9,3};
3
4     auto ret1 = std::ranges::remove(v1.begin(),
5                                     std::next(v1.begin(), 7), 3);
6     v1.erase(ret1.begin(), ret1.end()); // ret1이 subrange이므로 이 구간을 지우면
7     // 알아서 중간 부분이 지워진다.
8     show(v1); // 1,1,1,10,3,9,3
9 }
```

3.3.2 Algorithm vs member function

알고리즘의 `remove`와 멤버 함수의 `remove`가 있을 때 어떤 것을 사용하면 좋을까?

```

1 int main(){
2     std::list c{1,2,3,1,2,3};
3
4     c.erase(std::remove(c.begin(), c.end(), 3), c.end()); // 알고리즘 remove 사용
5     c.remove(3); // 멤버함수 remove 사용
6 }
```

-
- `std::list`는 별도의 최적으로 구현된 `remove` 멤버 함수가 존재한다.
 - 컨테이너에 알고리즘과 동일한 이름의 멤버 함수가 있다면 멤버 함수를 사용해라.

왜 컨테이너 안에 알고리즘과 동일한 이름의 멤버 함수가 있을까?

- 이유 1) 컨테이너의 반복자를 "알고리즘으로 보낼 수 없을 때"
- `std::sort` 알고리즘은 introsort(quick + heap) 기법을 사용하는데 "random access iterator"를 인자로 요구한다. 따라서 `list`의 반복자를 전달할 수 없다.
- `std::list` 안에는 다른 방식으로 구현된 `sort` 함수가 존재한다.

```
1 int main(){  
2     std::list c{1,2,3,1,2,3};  
3  
4     // #1  
5     std::sort(s.begin(), s.end());           // error  
6     std::ranges::sort(s.begin(), s.end()); // error  
7     std::ranges::sort(s);                  // error  
8     s.sort();                           // ok  
9 }
```

- 이유 2) 컨테이너의 "멤버 함수가 알고리즘보다 효율적일 때"
- `list`의 반복자도 `std::remove`에 전달할 수 있지만 `std::list`의 `remove`를 사용하는 것이 효율적이다.

```
1 int main(){  
2     std::list c{1,2,3,1,2,3};  
3  
4     // #2  
5     c.erase(std::remove(c.begin(), c.end(), 3), c.end()); // 알고리즘 remove 사용  
6     c.remove(3);                                         // 멤버함수 remove 사용  
7 }
```

컨테이너 안에 있는 모든 요소를 뒤집고 싶다!

- 1. 멤버 함수 중에서 `reverse()`가 있으면 사용
- 2. 멤버 함수가 없으면 알고리즘 `reverse()` 사용

```
1 int main(){  
2     std::list c{1,2,3,4,5};  
3     std::vector v{1,2,3,4,5};  
4  
5     std::ranges::reverse(v); // list, vector 모두 사용 가능  
6     s.reverse();           // list에만 존재 (권장)  
7 }
```

3.3.3 `std::erase`, `std::erase_if`

erase-remove idioms를 사용하면 컨테이너에서 원하는 요소를 지울 수 있지만 코드가 장황해보이는 단점이 있다. C++20부터는 `std::erase`, `std::erase_if` 알고리즘을 제공하여 편하게 원하는 요소를 제거할 수 있다.

```
1 int main(){  
2     std::list s{1,2,3,4,5};  
3     std::vector v{1,2,3,4,5};  
4  
5     auto cnt1 = std::erase(v, 3); // C++20부터 지원  
6     auto cnt2 = std::erase(s, 3); // C++20부터 지원  
7 }
```

- `std::erase()`: 리턴 값으로 제거된 요소의 개수를 반환한다
- 인자로는 컨테이너만 전달 가능하다 (반복자 불가능)

3.3.4 Algorithm with function as parameter

함수를 인자로 가지는 `std::transform`, `std::for_each` 알고리즘에 대해 살펴보자

```
1 int square(int a) { return a*a; }
2 int add(int a, int b) {return a+b; }
3 void print(int n) { std::print("{}\n", n); }

4
5 int main(){
6     std::vector v1{1,2,3,4,5};
7     std::vector<int> v2;
8     std::vector<int> v3;

9
10    std::ranges::transform(v1, std::back_inserter(v2), square); // v2: 1,4,9,16,25 v1을 제곱하여 v2에
11        추가한다
12    std::ranges::transform(v1, v2, std::back_insertter(v3), add); // v3: 2,6,12,20,30 v1과 v2를 더하여 v3에
13        추가한다
14    std::ranges::for_each(v3, print);                                // v3의 요소를 print를 통해 출력한다
15 }
```

- 단항 함수(unary function), 이항 함수(binary function)은 인자가 각각 1,2개인 함수를 말한다
- 각 알고리즘이 단항 함수를 요구하는지 이항 함수를 요구하는지 정확히 알아야 한다.

알고리즘에 전달되는 함수는 "지역 변수를 캡쳐할 수 있는" 장점을 가진다.

```
1 int main(){
2     std::vector v1{1,2,3,4,5};
3     std::vector<int> v2;
4
5     int k = 2;
6     std::ranges::transform(v1,
7         std::back_inserter(v2),
8         [k](int n) { return n + k; } ); // 지역 변수 캡쳐 가능
9
10
11 // int add_k(int n) { return n+k; } // 일반 함수를 사용하는 것은 권장되지 않는다.
```

3.3.5 Predicate

벡터 1,2,6,3,5에서 3의 배수를 찾고자 하는 경우를 보자

```
1 int main(){
2     std::vector v{1,2,6,3,5};
3
4     auto ret1 = std::ranges::find(v, 3);           // 벡터 내에서 3을 찾는다
5     auto ret2 = std::ranges::find_if(v,             // find_if를 사용하면 두번째 파라미터로
6         [] (int n) { return n%3 == 0; } );          // 단항함수(또는 조건자 predicate)를 넣을 수 있다
7     std::println("{}, {}", *ret1, *ret2);
8 }
```

- 조건자(predicate)는 bool 또는 bool로 변환 가능한 값을 반환하는 단항 함수를 말한다.
- `std::xxx_if` 형태로 구현된 함수에서 두번째 파라미터에 들어간다.

알고리즘의 조건자 버전 (`xxx_if`)

- `sort` 알고리즘은 `_if`를 사용하지 않는다.
- `find`도 `sort`처럼 조건자 버전의 이름을 동일하게 `find`로 하면 편하지 않았을까?
- c++98 시절의 문법으로는 인자의 갯수가 동일한 함수 템플릿을 같은 이름으로 여러개 만들 수는 없었다
(c++20 기술로는 concept을 사용하여 가능)

```
1 int main(){
2     std::vector v{1,2,6,3,5};
3 }
```

```

4 auto ret1 = std::ranges::find(v.begin(), v.end(), 3);
5 auto ret2 = std::ranges::find_if(v.begin(), v.end(),
6 [](int n) { return n%3 == 0; });
7
8 std::ranges::sort(v.begin(), v.end());
9 std::ranges::sort(v.begin(), v.end(), std::greater{}); // sort는 sort_if처럼 쓰지 않는다
10 }
```

std::predicate concept

- 조건자의 요구사항을 정의한 concept
- `std::predicate<T, Type>`: 단항 조건자 조사
- `std::predicate<T, Type1, Type2>`: 이항 조건자 조사

```

1 template<typename T>
2 void is_predicate(T f){
3     std::println("{}", std::predicate<T, int>);
4 }
5
6 int main(){
7     is_predicate([](int n) { return n%3 == 0; } ); // true
8     is_predicate([](int a, int b) { return a<b; } ); // false. 단항 조건자가 아니므로
9     is_predicate([](double a) { return a; } ); // true
10    is_predicate([](int n) { std::println("{}", n); } ); // false. 리턴값이 없으므로
11 }
```

3.3.6 Algorithm copy version

STL의 일부 알고리즘을 복사 버전을 제공한다.

- 알고리즘 이름이 "`_copy`"로 끝나는 알고리즘
- 연산의 수행 결과를 자신이 아닌 "다른 컨테이너에 저장"하는 알고리즘

```

1 int main(){
2     std::vector v1{1,2,3,1,2,3,1,2,3,1};
3     std::vector v2{0,0,0,0,0,0,0,0,0,0};
4
5     auto ret1 = std::ranges::remove(v1, 3);
6     auto ret2 = std::ranges::remove_copy(v1, v2.begin(), 3); // v1에서 3이 제거된 결과를 v2에 저장한다
7 }
```

- `std::remove`의 반환값: v1의 반복자(end)를 제공한다
- `std::remove_if`의 반환값: v3의 반복자(end)를 제공한다

c++20 constrained 알고리즘 복사버전의 반환값은 조금 복잡하다

```

1 int main(){
2     std::vector v1{1,2,3,1,2,3,1,2,3,1};
3     std::vector v2{1,2,3,1,2,3,1,2,3,1};
4     std::vector v3{0,0,0,0,0,0,0,0,0,0};
5
6     auto ret1 = std::ranges::remove(v1.begin(), std::next(v1.begin(),7), 3); // v1에서 처음 7개
7     범위에서 값을 지운다
8
9     auto ret2 = std::ranges::remove_copy(v2.begin(), std::next(v2.begin(),7), v3.begin(), 3); // v2에서
10    처음 7개 범위에서 값을 지우고 v3에 복사
11 }
```

- ret1의 타입은 `std::ranges::subranges` 타입이다
- ret1.begin(): 7개 범위 내에서 원소 제거 후 end()를 가르키는 반복자 - ret1.end(): 8번째 위치를 가르키는 반복자 (범위가 7이었으므로 그 다음 원소)
- ret2의 타입은 `std::ranges::remove_copy_result<I,O>` 타입이다 (`std::ranges::in_out_result<I,O>` 타입의 alias)

-
- ret2.in: v2의 8번째 원소를 가르키는 반복자
 - ret2.out: v3의 end를 가르키는 반복자

STL의 일부 알고리즘은 4가지 변형 버전을 제공한다 (기본형, _if, _copy, _copy_if)

```

1 int main(){
2     std::vector v1{1,3,6,3,5};
3     std::vector v2{0,0,0,0,0};
4
5     auto predicate = [](int n){ return n%3 == 0; }
6
7     // c++98
8     auto ret1 = std::remove(v1.begin(), v1.end(), 3);
9     auto ret2 = std::remove_if(v1.begin(), v1.end(), predicate);
10    auto ret3 = std::remove_copy(v1.begin(), v1.end(), v2.begin(), 3);
11    auto ret4 = std::remove_copy_if(v1.begin(), v1.end(), v2.begin(), predicate);
12
13    // c++20 constrained algorithm
14    auto ret5 = std::ranges::remove(v1, 3);
15    auto ret6 = std::ranges::remove_if(v1, predicate);
16    auto ret7 = std::ranges::remove_copy(v1, v2.begin(), 3);
17    auto ret8 = std::ranges::remove_copy_if(v1, v2.begin(), predicate);
18 }
```

3.3.7 Projection

c++20에서 추가된 projection에 대해 알아보자. Point 객체를 보관하는 컨테이너에서 y가 3인 요소를 찾고 싶다면 어떻게 할까?

```

1 struct Point {
2     int x,y;
3 }
4
5 int main(){
6     std::vector<Point> v{{1,1}, {2,2}, {3,3}, {4,4}};
7
8     auto ret1 = std::ranges::find_if(v,
9         [] (const Point& pt) { return pt.y == 3;}); // 조건자 사용
10    auto ret2 = std::ranges::find(v, 3,&Point::y); // projection 사용 (c++20)
11 }
```

- **Projection:** 컨테이너의 모든 요소를 한 개씩 projection에 통과시켜서 나오는 결과가 3인 것을 검색한다

```

1 struct Point {
2     int x,y;
3     int get_y() const { return y; }
4 }
5
6 int main(){
7     std::vector<Point> v{{1,1}, {2,2}, {3,3}, {4,4}};
8
9     // Projection
10    auto ret1 = std::ranges::find(v, 3, &Point::y); // 멤버 데이터 포인터
11    auto ret2 = std::ranges::find(v, 3, &Point::get_y()); // 멤버 함수 포인터
12    auto ret3 = std::ranges::find(v, 3, // 단항 함수
13        [] (const Point& p){ return p.y; });
14
15    // find_if
16    auto ret3 = std::ranges::find_if(v, 3, // find_if는 단항 조건자만 가능하다
17        [] (const Point& p){ return p.y == 3; });
18 }
```

- projection에 넣을 수 있는 것들은 (1) 멤버 데이터에 대한 포인터, (2) 멤버 함수에 대한 포인터, (3) 단항 함수 (함수 객체)가 들어갈 수 있다.

-
- 단항 함수와 단항 조건자(bool return)를 헷갈리지 않도록 유의한다!
 - 위 코드는 컨테이너 베전이지만 반복자 베전(v.begin(), v.end())도 가능하다.
 - find 뿐만 아니라 find_if에서도 가능하다

```

1 // find_if + projection
2 auto ret3 = std::ranges::find_if(v,
3 [](int n) { return n == 1; },
4 &Point::y);

```

- find, find_if 뿐만 아니라 대부분에 알고리즘에서도 projection을 지원한다

Projection을 사용하여 벡터 안에 있는 문자열을 사전 순서가 아닌 문자열의 길이를 기준으로 정렬해보자

```

1 int main(){
2     std::vector<string> v{"AAAA", "D", "BB", "CCC"};
3
4     std::ranges::sort(v);                                // 기본형 (사전 순서)
5
6     std::ranges::sort(v, [](std::string& s1, std::string& s2), // 조건자 사용 (길이 순서)
7     { return s1.size() < s2.size(); });
8
9     std::ranges::sort(v, {}, &std::string::size);          // Projection 사용. 이항 함수는 기본형이 less
10    이므로 를 전달하면 less가 호출됨
11
12    std::ranges::sort(v, std::greater{}, &std::string::size); // Projection 사용. greater 버전
}

```

3.3.8 Constrained algorithm function object

c++20에 추가된 constrained algorithm이 함수 객체라는 사실에 대해 자세히 알아보자.

c++98	<code>std</code> 이름 공간 인자로 반복자만 사용 가능 함수(템플릿)으로 구현
c++20	<code>std::ranges</code> 이름 공간 인자로 반복자 뿐만 아니라 컨테이너도 전달 가능 보다 많은 정보를 담은 반환값 함수가 아닌 함수 객체로 구현

```

1 int main(){
2     auto ret1 = std::max(1, 2);                      // 함수
3     auto ret2 = std::ranges::max(1, 2);                // 함수 객체
4     auto ret3 = std::ranges::max.operator()(1, 2); // 함수 객체
5 }

```

Argument Dependent Lookup (ADL)

- 함수를 찾을 때 인자가 속해 있는 이름 공간은 자동으로 검색하도록 하는 기법

```

1 namespace Graphics {
2     struct Point {
3         int x,y;
4     };
5     void draw_pixel(const Point& pt) {}
6 };
7
8 int main(){
9     Graphics::Point pt{1,2};
10    Graphics::draw_pixel(pt);
11    draw_pixel(pt);           // ok. 이름 공간을 명시하지 않았으므로 안될 것 같으나 ADL에 의해 가능하다
12 }

```

- pt는 Graphics 이름 공간 안에 있으므로 `draw_pixel`도 같은 Graphics 이름 공간 안에 있는지 자동으로 검색한다.

ADL이 필요한 이유: 특정 이름 공간 안에 있는 객체 등에 연산자 재정의(operator overloading) 문법 등을 사용하려면 ADL 문법이 필요하다.

```
1 namespace Graphics {
2     struct Point {
3         int x,y;
4     }
5
6     Point operator+(const Point& p1,
7         const Point& p2) {
8         return Point{p1.x + p2.x, p1.y + p2.y};
9     }
10 };
11
12 int main(){
13     Graphics::Point p1{1,1};
14     Graphics::Point p1{2,2};
15
16     auto ret1 = Graphics::operator+(p1, p2); // 보통 이렇게 작성하지 않는다
17     auto ret2 = p1 + p2; // operator+(p1,p2)가 호출되는데 ADL이 없으면 컴파일 에러가
18     발생한다
19 }
```

```
1 int main(){
2     int n1 = 10;
3     int n2 = 20;
4
5     std::string s1 = "AA";
6     std::string s2 = "BB";
7
8     auto ret1 = std::max(n1, n2); // ok
9     auto ret2 = std::max(n1, n2); // ok
10    auto ret3 = max(n1, n2); // error
11    auto ret4 = max(n1, n2); // ok. string이 std 이름 공간 안에 있으므로 ADL이 적용됨
12 }
```

ADL로 인해 컴파일 에러가 발생하는 예시를 살펴보자

```
1 namespace mystd {
2     class string {};
3
4     template<typename T>
5     void max(const T& a, const T& b) { std::println("std::max"); }
6
7     namespace ranges {
8         template<typename T>
9         void max(const T& a, const T& b) { std::println("std::ranges::max"); }
10    }
11 }
12
13 int main(){
14     mystd::string s1, s2;
15
16     mystd::max(s1, s2); // ok
17     mystd::ranges::max(s1, s2); // ok
18
19     using namespace mystd::ranges; // 신버전 이름 공간을 열어놓은 경우를 살펴보자
20     max(s1, s2); // error! ADL 이름 공간과 열어놓은 mystd::ranges 이름 공간이 같은 이름의
21     함수로 충돌한다
22 }
```

- 위와 같은 ADL 이름 충돌을 막기 위해 c++20에서는 constrained algorithm들을 **함수가 아닌 함수 객체로 구현**하였다.

```

1 namespace mystd {
2     class string {};
3
4     template<typename T>
5     void max(const T& a, const T& b) { std::println("std::max"); }
6
7     namespace ranges {
8         struct max_fn {
9             template<typename T>
10            void operator()(const T& a, const T& b) { std::println("std::ranges::max"); }
11        };
12        max_fn max;           // 함수 객체로 구현되었다
13    }
14 }
```

3.4 Container

3.4.1 Container basic

STL에는 다양한 자료구조를 구현한 클래스 템플릿들이 존재한다.

2*sequence container	C++98	vector, list, deque
	C++11	array, forward_list
associative container	C++98	set, multiset, map, multimap
unordered associative container	C++11	unordered_set, unordered_multiset, unordered_map, unordered_multimap
2*container adapter	C++98	stack, queue, priority_queue
	C++23	flat_set, flat_multiset, flat_map, flat_multimap
2*view	C++20	span
	C++23	mdspan

- 대부분의 컨테이너는 "공통적인 특징"이 있으므로 종류별로 사용법이 유사하다.
- 각 컨테이너가 사용하는 "자료구조의 특징(linked list, tree, hash 등)을 이해하는 것이 중요"하다.

STL 컨테이너의 공통적인 특징 첫 번째는 모든 STL 컨테이너는 **member type**이 존재한다.

```

1 int main() {
2     std::vector v{1,2,3,4};
3
4     auto sz1 = v.size();           // 권장
5     std::vector<int>::size_type sz2 = v.size(); // 가장 정확한 코드
6 }
```

- `v.size()`의 반환 타입은 어떤 타입일까?
- 이는 STL의 실제 구현 환경에 따라 다를 수 있다
- C++ 표준에서 각 컨테이너는 "size_type"이라는 member type이 있어야 하고 `size()` 함수는 `size_type`을 반환해야 한다

```

1 template<typename T,
2 typename Alloc = std::allocator<T>>
3 class vector {
4 public:
5     // ...
6     using value_type = T;
7     using size_type = std::size_t;
8
9     size_type size() const;
10    // ...
11};
```

`size_type` 이외에도 다양한 member type들이 존재한다.

<code>value_type</code>	<code>T</code>
<code>allocator_type</code>	<code>Allocator</code>
<code>size_type</code>	usually <code>std::size_t</code>
<code>difference_type</code>	usually <code>std::ptrdiff_t</code>
<code>reference</code>	<code>value_type&</code>
<code>const_reference</code>	<code>const value_type&</code>
<code>pointer</code>	...
<code>const_pointer</code>	...
<code>iterator</code>	...
<code>const_iterator</code>	...
<code>reverse_iterator</code>	...
<code>const_reverse_iterator</code>	...

두 번째 STL 컨테이너의 공통적인 특징은 컨테이너가 가진 대부분의 멤버 함수는 "반환과 제거를 동시에 하지 않는다"는 것이다.

```
1 int main(){
2     std::vector v{1,2,3,4};
3
4     auto n1 = v.pop_back();      // error! pop_back은 아무것도 반환하지 않는다
5     auto n2 = v.back();
6     v.pop_back();
7
8     std::stack<int> s;
9     s.push(10);
10    s.push(20);
11    std::println("{}", s.top()); // 20
12    std::println("{}", s.top()); // 20. 위 코드에서 반환만하지 제거되지 않으므로 똑같은 값이 나온다. 중간에
13        s.pop()을 넣어줘야 맨 위 값이 제거된다
}
```

- `v.back()`: 마지막 요소를 반환(reference)하고 요소는 제거되지 않는다
- `v.pop_back()`: 마지막 요소를 제거만 하고 반환하지 않는다
- **예외 안정성(exception safety)의 강력 보장**을 위해 일부러 반환과 제거를 분리하여 설계하였다. 또한 값 반환이 아닌 참조(reference) 반환하여 임시 객체를 제거하도록 설계하였다

3.4.2 Allocator

세 번째 STL 컨테이너의 공통적인 특징은 **단위 전략 디자인(Policy base design)**을 많이 사용한다. 예를 들면 allocator, compare, hash 등이 있다. 이 중 allocator에 대해 자세히 알아보자.

```
1 template<typename T>
2 class vector {
3     T* ptr;
4
5     public:
6         using size_type = std::size_t;
7
8     void resize(size_type sz) {
9         // 새롭게 메모리를 할당해야 한다면 어떤 방법을 사용하는 것이 좋을까?
10    }
11 };
12
13 int main(){
14     vector<int> v{1,2,3,4};
15     v.resize(8);
16 }
```

- 얼핏 생각하면 `new`, `delete`를 사용하면 될 것 같지만 C++에서 메모리를 할당, 해제하는 방법은 다양하다
- 1) `new`, `delete`
- 2) `malloc`, `free`

-
- 3) 각 OS가 제공하는 system call
 - 4) memory pooling 기술 사용

만약 vector의 `resize()` 멤버 함수에서 `new`, `delete`를 직접 사용했다면 vector 사용자가 다른 방식으로 변경할 수 없다. 따라서 STL 컨테이너는 사용자가 메모리의 할당, 해지 방식을 교체할 수 있도록 "allocator"라는 개념을 사용한다.

- 컨테이너가 내부적으로 메모리를 할당, 해지하는 방법을 사용자가 변경할 수 있도록 해보자

```

1 template<typename T, typename Alloc = std::allocator>
2 class vector {
3     T* ptr;
4     Alloc ax;
5
6     public:
7     using size_type = std::size_t;
8
9     void resize(size_type sz) {
10         ptr = ax.allocate(sz); // 두 함수를 반드시 구현해야 한다
11         ax.deallocate(ptr, sz);
12     }
13 };
14
15 int main(){
16     vector<int> v{1,2,3,4};
17     v.resize(8);
18 }
```

- 위와 같이 클래스가 사용하는 다양한 정책을 템플릿 인자로 변경할 수 있게 하는 디자인을 **단위 전략 디자인 (Policy base design)**이라고 한다
- STL에서 많이 사용하는 기법이다
- allocator: 메모리 할당, 해지를 추상화한 클래스이다. 사용자가 만들 수도 있고 C++ 표준이 제공하기도 한다
- `std::allocator`: C++ 표준에서 제공하는 메모리 할당기. 내부적으로는 `operator new()`, `operator delete()`를 사용한다

allocator를 직접 만들어보자.

```

1 template<typename T>
2 class debug_alloc{
3     public:
4     using value_type = T;
5     debug_alloc() {}
6     template<typename U> debug_alloc(debug_alloc<U>&) {}
7
8     T* allocate(std::size_t sz) {
9         void* ptr = malloc(sizeof(T) * sz);
10        std::println("allocate: {}, {} cnts", static_cast<void*>(ptr), sz);
11        return static_cast<T*>(ptr);
12    }
13
14    void deallocate(T* ptr, std::size_t sz) {
15        std::println("deallocate: {} {} cnts", static_cast<void*>(ptr), sz);
16        free(ptr);
17    }
18 };
19
20 template<typename T>
21 bool operator==(const debug_alloc<T>& a1, const debug_alloc<T>& a2) { return true; }
22
23 template<typename T>
24 bool operator!=(const debug_alloc<T>& a1, const debug_alloc<T>& a2) { return false; }
25
26 int main(){
27     std::vector<int, debug_alloc<int>> v{1,2,3,4};
28 }
```

```

29 std::println("=====");
30 v.resize(8);
31 std::println("=====");
32 }

```

- allocator가 지켜야 되는 요구조건은 상당히 복잡하지만 대부분 optional로 반드시 구현하지 않아도 된다. 자세한 내용은 cppreference를 참조하면 된다

3.4.3 Sequence container

Sequence 컨테이너란 요소들이 **삽입된 순서를 유지하면서 한 줄(선형)으로 놓여 있는 컨테이너**를 말한다. Sequence 컨테이너의 종류와 이들의 카테고리는 다음과 같다

<code>forward_list</code>	forward iterator
<code>list</code>	bidirectional iterator
<code>deque</code>	random access iterator
<code>vector</code>	contiguous iterator
<code>array</code>	contiguous iterator

Sequence 컨테이너의 템플릿 파라미터는 다음과 같다.

```

1 template<typename T, typealias Allocator = std::allocator<T>> class forward_list;
2 template<typename T, typealias Allocator = std::allocator<T>> class list;
3 template<typename T, typealias Allocator = std::allocator<T>> class deque;
4 template<typename T, typealias Allocator = std::allocator<T>> class vector;
5 template<typename T, std::size_t N> class array; // array만 다르다

```

Sequence 컨테이너의 **앞쪽**에 데이터를 추가, 제거, 접근하려면 `push_front`, `pop_front`, `front`, `emplace_front`를 사용하면 되고 **뒤쪽**에 데이터를 추가, 제거, 접근하려면 `push_back`, `pop_back`, `back`, `emplace_back`을 사용하면 된다. 데이터 **중간**에는 `insert`, `erase`, `emplace`를 사용하여 추가, 제거, 접근할 수 있다.

```

1 int main{
2     std::list s{1,2,3,4,5};
3     std::deque d{1,2,3,4,5};
4     std::vector v{1,2,3,4,5};
5
6     // #1. 사용법이 유사하다
7     s.push_back(10);
8     d.push_back(10);
9     v.push_back(10);
10
11    // #2. vector는 앞쪽으로 추가, 제거 할 수 없다
12    s.push_front();
13    d.push_front();
14    v.push_front(); // error!
15
16    // #3. [] 연산은 deque와 vector만 가능하다
17    s[0] = 10;      // error!
18    d[0] = 10;
19    v[0] = 10;
20 }

```

3.4.4 `std::vector`

`std::vector`를 생성하는 방법에 대해 알아보자

```

1 int main() {
2     std::vector v1;          // error!
3     std::vector<int> v2;    // ok
4
5     // {}, ()의 초기화 방식이 다르다!
6     std::vector v3{5,2};    // [5,2] 2개 원소 초기화
7     std::vector v4(5,2);    // [2,2,2,2,2] 5개의 2로 초기화
8

```

```

9 std::vector v5{5};      // [5] 값으로 초기화
10 std::vector<int> v6(5); // [0,0,0,0,0] 5개의 0으로 초기화
11
12 std::vector v7(v3);                // [5,2]
13 std::vector v8(v4.begin(), std::next(v4.begin(),3)); // [2,2,2]
14 }
```

다음으로 `std::vector` 요소의 삽입, 삭제에 대해 알아보자

```

1 int main() {
2 std::vector v1{0,0,0};
3 std::vector v2{1,2,3};
4
5 // 삽입: [0,0,0]에서 시작
6 v1.push_front(0);           // error!
7 v1.push_back(4);           // [0,0,0,4]
8 v1.insert(v1.begin(), 9);   // [9,0,0,0,4]
9 v1.insert(v1.end(), v2.begin(), v2.end()); // [9,0,0,0,4,1,2,3]
10 v1.insert_range(v1.end(), v2);    // 위 코드와 동일 [9,0,0,0,4,1,2,3,1,2,3]
11 v1.append_range(v2);          // 위 코드와 동일 [9,0,0,0,4,1,2,3,1,2,3,1,2,3]
12
13 // 삭제
14 v1.pop_front();           // error!
15 v1.pop_back();            // [9,0,0,0,4,1,2,3,1,2,3,1,2]
16 v1.erase( v1.begin() );   // [0,0,0,4,1,2,3,1,2,3,,1,2]
17 v1.erase( v1.begin(), std::next(v1.begin(), 3)); // [4,1,2,3,1,2,3,1,2]
18
19 v1.clear();               // []
20 }
```

- `insert_range`, `append_range`는 C++23에서 추가되었다.

`std::vector` 요소에 접근하는 코드를 살펴보자

```

1 int main() {
2 std::vector v{1,2,3,4};
3
4 v.front() = 0;
5 v.back() = 0;
6 v[2] = 0;
7 v.at() = 0;
8
9 try {
10     v.at(20) = 0;    // 예외 발생
11     v[20] = 0;       // undefined behavior로 프로그램 종료
12 }
13 catch(std::exception& e) {
14     std::println( "catch : {}", e.what() )
15 }
16
17 }
```

- `v.front()`: 첫번째 요소를 참조로 반환. 제거 안됨
- `v.back()`: 마지막 요소를 참조로 반환. 제거 안됨
- `v.at(idx)`: idx 번째 요소를 참조로 반환. 잘못된 idx 전달 시 예외 발생
- `v[idx]`: idx 번째 요소를 참조로 반환. 잘못된 idx 전달 시 undefined behavior로 프로그램 종료. 반드시 유효한 idx를 전달해야 한다.

`v.at(idx)`와 `v[idx]` 중 어떻게 접근하는 방법이 더 좋을까?

```

1 int main() {
2 std::vector v{1,2,3,4};
3
4 auto sz = v.size();
5 for(int i=0; i<sz; i++){
```

```

6     v[i] = 0;      // 예외가 발생할 확률이 없기 때문에 더 빠르다
7     v.at(i) = 0;   // 내부적으로 i가 유효한지 검사하므로 느린편이다
8 }
9 }
```

`std::vector`의 요소를 함수 인자로 받으려면 어떻게 보내는게 좋을까?

```

1 void fn(int* p, int sz) {
2 }
3
4 int main() {
5     int x[4] = {1,2,3,4};
6     fn(x, 4);           // ok
7
8     std::vector v{1,2,3,4};
9     fn(v, v.size());    // error
10    fn(&v, v.size());   // error
11    fn(v[0], v.size()); // ok
12    fn(v.data(), v.size()); // ok
13 }
```

- `v.data()`를 사용하면 벡터 타입의 포인터를 반환하기 때문에 이를 통해 함수를 호출할 수 있다

두 벡터를 바꾸려면 어떻게 해야할까?

```

1 int main() {
2     std::vector v1{1,2,3,4};
3     std::vector v1{5,6,7};
4
5     v1.swap(v2);
6     // v1: [5,6,7]
7     // v2: [1,2,3,4]
8
9     v1.assign(v2.begin(), v2.end()); // v1의 모든 요소를 v2의 요소로 할당하라
10    // v1: [1,2,3,4]
11    // v2: [1,2,3,4]
12 }
```

3.4.5 Capacity

벡터의 크기와 용량(capacity)라는 개념에 대해 알아보자

```

1 int main(){
2     std::vector<int> v(10);
3     v.resize(7);           // 어떻게 동작할까?
4 }
```

- 벡터의 크기를 10개에서 7개로 줄이는 경우 어떻게 동작할까?
- 1) 7개의 크기를 재할당할 것이다
- 2) 10개 크기의 기존 메모리를 계속 사용할 것이다
- 정답은 2번이다
- 이 때 `size`는 7이지만 `capacity`는 10의 값을 가짐으로써 벡터의 용량(capacity)를 별도로 표기한다

size	벡터 요소의 개수
capacity	벡터가 실제 사용하는 메모리 크기

```

1 int main(){
2     std::vector<int> v(10);
3     v.resize(7);
4     std::println("{} , {}", v.size(), v.capacity()); // 7,10
5 }
```

위 상황에서 값을 넣어보자

```
1 int main(){
2     std::vector<int> v(10);
3     v.resize(7);
4
5     std::println("{}, {}", v.size(), v.capacity()); // 7,10
6
7     v.push_back(0); // 8,10
8
9     v.shrink_to_fit(); // 8,8 : capacity를 size에 맞춘다
10
11    v.push_back(0); // 9,? : 메모리 재할당이 필요하다
12
13    v.resize(17); // 17,? : 메모리 재할당이 필요하다
14 }
```

- capacity가 커져야 할 경우 컴파일러마다 증가값이 다르다.

```
1 int main(){
2     std::vector<int> v1(10);
3     std::vector<int> v2;
4     v2.reserve(10); // size와 관계없이 capacity만 확보한다
5
6     v1.push_back(0); // 11,xx : 10,10에서 값이 들어오므로 capacity가 같이 늘어난다
7     v2.push_back(0); // 1,10 : 0,10에서 값이 들어오므로 1,10이 된다.
8 }
```

capacity와 생성자, 소멸자의 관계에 대해 살펴보자

```
1 struct Object{
2     Object() { std::println("{} {}", __func__); }
3     ~Object() { std::println("{} {}", __func__); }
4 };
5
6 int main(){
7     std::vector<Object> v1(5); // 생성자 5회 호출
8
9     v1.resize(3); // 메모리는 제거되지 않지만 소멸자는 호출되어야 한다. 소멸자 2회 호출.
10
11    v1.resize(4); // 생성자 1회 호출
12    v1.push_back(Object{}); // 생성자 1회 호출
13
14    std::vector<Object> v2;
15    v2.reserve(5); // 메모리만 잡아놓으므로 아무것도 호출되지 않음
16
17    v2.resize(1); // 생성자 1회 호출
18 }
```

- resize와 capacity에 관계없이 값이 벡터가 삽입, 삭제될 때마다 생성자, 소멸자는 계속 호출된다

벡터의 복사와 이동(move)는 capacity와 어떤 관계를 가질까?

```
1 int main(){
2     std::vector v1(5,0);
3     v1.resize(3); // 3,5
4
5     std::vector v2 = v1; // v1: 3,5, v2: 3,3 : 벡터의 사이즈만큼만 복사한다
6
7     std::vector v3 = std::move(v1); // v1: 0,0, v3: 3,5 : 벡터를 그대로 이동하므로 똑같은 사이즈와 capacity를 갖는다
8 }
```

사이즈와 capacity를 맞추고 싶을 땐 shrink_to_fit을 사용하면 된다. 하지만 이는 Modern C++에서 나온 개념이므로 이전에는 다른 방법을 사용하여 둘을 맞추었다.

```

1 int main() {
2     std::vector v(5,0);
3     v.resize(3);
4
5     v.shrink_to_fit();           // new version
6
7     std::vector<int>(v).swap(v); // old version
8 }
```

- `std::vector<int>(v)`: 복사 생성자를 사용한 임시 객체 생성
- `.swap(v)`: 임시 객체와 `v`를 바꿈으로써 `size`와 `capacity` 크기를 맞춤

벡터의 다양한 연산들과 `capacity`의 관계에 대해 살펴보자

```

1 int main(){
2     std::vector v = {1,2,3,1,2,3,1,2,3}; // 9,9
3
4     auto ret = std::ranges::remove(v, 3); // 9,9 : 3을 뒤로 보내기만 하므로 사이즈, 메모리 변경 없음
5
6     v.erase(ret.begin(), ret.end());      // 6,9
7
8     v.clear();                         // 0,9
9
10    v.shrink_to_fit();                // 0,0
11 }
```

- `erase`, `clear`를 해도 실제 메모리는 제거되지 않는다. `shrink_to_fit`을 호출해야 실제 메모리가 제거된다.

이번에는 벡터가 아닌 다른 컨테이너에 대해 살펴보자

```

1 int main(){
2     std::list s{1,2,3};
3     std::deque d{1,2,3};
4     std::vector v{1,2,3};
5     std::string str{"hello"};
6
7     std::println("{}", s.capacity()); // error!
8     std::println("{}", d.capacity()); // error!
9     std::println("{}", v.capacity());
10    std::println("{}", str.capacity());
11
12    s.shrink_to_fit();             // error!
13    d.shrink_to_fit();
14    v.shrink_to_fit();
15    str.shrink_to_fit();
16 }
```

- `capacity`는 연속된 메모리(또는 연속된 메모리와 유사한 `deque` 등)를 사용하는 컨테이너에만 존재하는 개념이다. 따라서 `list`, `deque`에는 `capacity` 개념이 존재하지 않는다.

3.4.6 Container & user define type

컨테이너에 사용자 정의 타입을 보관하는 것에 대해 알아보자

```

1 class Point {
2     int x,y;
3 public:
4     Point(int a, int b) : x{a}, y{b} {} // 디폴트 생성자가 없다
5 };
6
7 int main() {
8     std::vector<Point> v1;           // ok
9     std::vector<Point> v2(4);        // error! 초기 4개만큼 크기를 확보할 때 디폴트 생성자를 불러야
10    하는데 없기 때문에 에러 발생
```

```

10 std::vector<Point> v3(4, Point{0,0}); // ok. 복사 생성자를 사용하여 초기화
11
12 v3.resize(8); // error! 크기를 변경할 때 디폴트 생성자가 필요함
13 v3.resize(8, Point{0,0}); // ok
14 }

```

- **디폴트 생성자가 없는 타입을 저장하는 경우** `resize()` 등 멤버 함수 사용 시 새로 추가되는 요소를 초기화하기 위한 견본 객체를 전달해야 한다.

```

1 class Point {
2 public:
3     int x,y;
4
5 public:
6     Point(int a, int b) : x{a}, y{b} {}
7
8     void print() const { std::println("{}, {}", x,y); }
9
10    auto operator<=>(const Point&) const = default; // (c++20) 컴파일러가 자동으로 멤버 변수 순으로 <,> 연산을 수행한다.
11 };
12
13 int main() {
14     std::vector<Point> v{{0,0},{3,3},{2,2},{1,1}};
15
16     // #1
17     std::sort(v.begin(), v.end());
18
19     // #2
20     std::sort(v.begin(), v.end(), [] (const Point& p1, const Point& p2){ return p1.x < p2.x; });
21 }

```

- `std::vector<user type>`에 알고리즘을 적용하려면 user type은 알고리즘이 요구하는 연산을 수행할 수 있어야 한다.

- `sort()` 알고리즘은 < 연산으로 비교하기 때문에 user type이 < 연산이 가능해야 한다.

- `sort()`를 사용하는 방법은 user type에 연산자(<,=,>) 함수를 작성하거나 lambda 함수를 사용하여 비교할 수 있다.

3.4.7 Emplace

컨테이너들이 가지고 있는 `emplace_XXX` 멤버 함수들에 대해 알아보자.

우선 내부적으로 메모리를 할당해서 데이터를 관리하는 `Buffer` 클래스를 만들어보자.

```

1 class Buffer {
2     std::string desc;
3     std::size_t sz;
4     char* ptr;
5
6     public:
7     Buffer(const std::string& desc, std::size_t size)
8         : desc{desc}, sz{size}, ptr{new char[size]} {
9         std::println("Buffer Constructor");
10    }
11
12    Buffer(std::string&& desc, std::size_t size)
13        : desc{std::move(desc)}, sz{size}, ptr{new char[size]} {
14        std::println("Buffer Constructor");
15    }
16
17    ~Buffer() {
18        if(ptr) delete[] ptr;
19        std::println("Buffer Destructor");
20    }
21
22    Buffer(const Buffer& other)

```

```

23 : desc{ other.desc }, sz{ other.sz }
24 {
25     ptr = new char[sz];
26     memcpy(ptr, other.ptr, sz);
27
28     std::println("Buffer Copy Constructor");
29 }
30
31 Buffer(Buffer&& other) noexcept
32 : desc{ std::move(other.desc) }, sz{ other.sz }, ptr{other.ptr}
33 {
34     other.ptr = nullptr;
35     other.sz = 0;
36     std::println("Buffer Move Constructor");
37 }
38
39 int size() const { return sz; }
40
41 Buffer& operator=(const Buffer&) = delete;
42 Buffer& operator=(Buffer&&) = delete;
43 };

```

- 생성자, 소멸자, 복사 생성자, 이동(move) 생성자를 제공하고 각각 언제 호출되는지 확인하기 위해 화면에 log를 출력한다.

```

1 int main(){
2     Buffer b1{"mybuffer1", 4096};
3     Buffer b2{"mybuffer2", 4096};
4
5     Buffer b3 = b1;           // 복사 생성자 호출
6     Buffer b4 = std::move(b2); // 이동 생성자 호출
7
8     std::println("{}," , b1.size()); // 4096
9     std::println("{}," , b2.size()); // 0
10 }

```

- b1은 깊은 복사를 통해 b3에 복사되므로 자체의 크기는 변하지 않는다. 하지만 b2는 이동 생성자를 통해 b4에 전달되므로 크기가 0이 된다.

```

1 int main(){
2     std::vector<Buffer> v;
3
4     Buffer b{"mybuffer", 4096};           // 생성자 호출
5
6     v.push_back(b);                     // #1
7     v.push_back(std::move(b));          // #2
8     v.push_back(Buffer{"mybuffer", 4096}); // #3
9     v.emplace_back("mybuffer", 4096);    // #4
10
11 std::println("=====");
12 }

```

- #1: b의 복사본을 만들 때 복사 생성자 호출. 프로그램이 끝나면 b 파괴, 벡터 안에 있는 요소도 파괴되므로 소멸자 2번 호출 (생성 2회, 소멸 2회)
- #2: b가 이동하면서 이동 생성자 호출. 프로그램이 끝나면 b 파괴, 벡터 안에 있는 요소도 파괴되므로 소멸자 2번 호출 (생성 2회, 소멸 2회)
- #3: 임시 객체가 이동하면서 이동 생성자 호출. 해당 라인을 넘어가자마자 임시 객체는 소멸됨. 프로그램이 끝나면 벡터 요소가 파괴되며 소멸자 1번 호출 (생성 2회, 소멸 2회)
- #4: #1~#3 방법과 다르게 객체를 생성하지 않고 바로 컨테이너에 넣는 방법. 생성자 1번 호출. 프로그램 끝나면 소멸자 1번 호출 (생성 1회, 소멸 1회)

벡터에서 push_back과 emplace_back의 구현의 차이점을 살펴보자

```

1 template<typename T, typename Alloc = std::allocator<T>>
2 class vector{
3 public:
4 // 함수 인자로 "요소와 같은 타입의 객체를 한 개 받아서
5 // 복사본 객체를 생성해서 보관
6 void push_back(const T& obj) { new T{obj}; }
7 void push_back(T&& obj) { new T{std::move(obj)}; }
8
9 // 함수 인자로 "임의 타입의 임의 갯수를 받아서
10 // 객체 생성
11 template<typename ... ARGS>
12 decltype(auto) emplace_back(ARGS&& ... args) {
13     new T(std::forward<ARGS>(args)...);
14     // ...
15 }
16 };

```

`push_back`과 `emplace_back`의 차이점을 표로 나타내면 다음과 같다.

<code>push_back()</code>	요소(<code>T</code>)와 같은 타입의 객체 한 개를 받아서 복사본을 생성하여 보관
<code>emplace_back()</code>	요소를 만들 때 생성자 인자로 사용할 임의 타입의 객체를 임의 갯수만큼 받아서 객체를 생성

```

1 class Point {
2 public:
3 Point(int a) {}
4 Point(int a, int b) {}
5 Point(const Point&) {}
6 };
7
8 int main(){
9     std::vector<Point> v;
10
11    v.push_back(1,2);      // error. push_back은 Point 타입만 받는다
12    v.emplace_back(1,2);  // ok. Point(int a, int b)
13
14    Point pt(1,2);
15    v.push_back(pt);     // ok. Point(const Point&)
16    v.emplace_back(pt);  // ok. Point(const Point&)
17
18    v.push_back(1);       // ok. 1로부터 Point 임시객체를 생성하여 push_back(임시객체)가 불려짐
19    v.emplace_back(1);   // ok
20 }

```

- `v.push_back(1)`의 경우 1로부터 임시 객체를 생성하여 `push_back(- 128) * 64 + (' - 128) 뼘 - 128) * 64 + (' - 128) 뼘` 가 호출된다. 만약 `explicit Point(int a)`와 같이 정의했다면 에러가 발생한다.

또 다른 예제를 보자

```

1 int main(){
2     std::vector<Buffer> v1;
3     v1.push_back(Buffer{"mybuffer", 4096}); // bad
4     v1.emplace_back("mybuffer", 4096);     // good
5
6     std::vector<int> v2;
7     v2.push_back(1);                      // good. more readable
8     v2.emplace_back(1);                  // so good
9 }

```

- 컨테이너에 primitive type을 보관하는 경우 두 멤버 함수의 구현은 거의 동일하다. 일부 가독성을 위해 `push_back` 이 더 낫다는 의견이 있다.

- 컨테이너에 user type을 보관하는 경우 `push_xxx`보다 `emplace_xxx`가 좋다.

3.4.8 std::array

std::array라는 컨테이너에 대해 살펴보자

```
1 int main() {
2     int x[5]{1,2,3,4,5};
3
4     std::vector v{1,2,3,4,5};
5
6     x[0] = 10;
7     v[0] = 10;
8 }
```

- 배열과 벡터의 차이점은 둘을 지역 변수로 생성 시 놓이는 위치가 다르다. 배열은 모든 요소가 스택에 보관되는 반면에 std::vector는 모든 요소를 힙에 보관한다.
- std::vector가 사용하기 편리하고 안전하지만 메모리 할당 속도, 컴파일러 최적화 등을 고려하면 배열이 가장 빠르다.

```
1 constexpr std::size_t cnt = 1000000;
2
3 void f1() {
4     for(int i=0; i<cnt; i++) {
5         std::vector<int> v{1,2,3,4,5};
6     }
7 }
8
9 void f2() {
10    for(int i=0; i<cnt; i++) {
11        int x[5]{1,2,3,4,5};
12    }
13 }
14
15 int main(){
16     chronometry(f1);
17     chronometry(f2);
18 }
```

- 두 함수의 실행 시간을 테스트해보면 약 100배 넘는 시간 차이가 난다. 벡터를 생성, 소멸하는 과정에서 많은 시간이 소요된다.

std::array는 모든 요소를 스택에 보관하는 컨테이너이며 c++11에 도입되었다. 우선 다음과 같은 간단한 예제를 살펴보자

```
1 template<typename T, int N>
2 struct array {
3     T buff[N];
4
5     int size() const { return N; }
6 };
7
8 int main() {
9     array<int, 5> a{1,2,3,4,5};           // buff[5]를 이와 같이 초기화 가능하다
10    // 멤버 변수가 배열이므로 스택에 값이 저장된다.
11    std::println("{}", sizeof(a));
12    std::println("{}", a.size());          // 멤버 함수 호출도 가능하다.
13 }
```

- std::array는 모든 요소를 스택에 보관하는 컨테이너이며 실제 배열과 거의 동일한 성능을 가진다. 또한 다양한 멤버 함수가 있어서 실제 배열보다 편리하고 안전하다.

```
1 constexpr std::size_t cnt = 1000000;
2
3 void f1() {
```

```

4  for(int i=0; i<cnt; i++) {
5      std::array<int, 5> v{1,2,3,4,5};
6  }
7 }

8
9 void f2() {
10 for(int i=0; i<cnt; i++) {
11     int x[5]{1,2,3,4,5};
12 }
13 }

14
15 int main(){
16 chronometry(f1);
17 chronometry(f2);
18 }
```

- 두 함수의 시간을 비교해보면 거의 동일하게 나온다. 즉 실제 배열과 성능이 동일하다

`std::array`의 특징을 살펴보자

```

1 int main() {
2     std::array<int, 5> a{1,2,3,4,5};
3
4 // #1 요소 접근
5 a[0] = 10;
6 a.at(0) = 20;
7 a.back() = 10;
8 a.front() = 10;
9
10 // #2 반복자
11 auto it1 = a.begin();
12 auto it2 = a.end();
13
14 // #3 삽입, 삭제, 크기 변경을 할 수 없음
15 a.push_back(1);    // error
16 a.pop_back();     // error
17 a.resize(10);     // error
18
19 // #4
20 auto r1 = a.size();
21 auto r2 = a.empty();
22
23 // #5 다른 array를 만들 때 복사 생성자 사용 가능
24 int x1[3]{1,2,3};
25 int x2[3] = x1;           // error. 배열은 이런 문법이 불가능하다
26
27 std::array<int,3> a1{1,2,3};
28 std::array<int,3> a2 = a1; // ok
29
30 int *px = x1;            // ok. 실제 배열은 포인터로 암시적 형변환 된다
31 int *pa = a1;             // error. array는 포인터로 변경 불가능하다
32 int *pa = a1.data();      // ok
33 }
```

- `std::array`는 삽입, 삭제, 크기 변경을 할 수 없다

3.4.9 `std::move` (`std::array`)

`std::array`를 이동했을 때 효과에 대해 살펴보자.

```

1 int main() {
2     std::array<int,3> a1{1,2,3};
3     std::array<int,3> a2{1,2,3};
4
5     std::array<int,3> a3 = a1;           // 깊은 복사
```

```

6 std::array<int,3> a4 = std::move(a2); // int는 이동을 지원하지 않으므로 위와 동일하게 복사를 수행한다
7
8
9 std::array<string,3> a5{"AA","BB","CC"};
10 std::array<string,3> a6{"AA","BB","CC"};
11
12 std::array<string,3> a7 = a5;           // 깊은 복사
13 std::array<string,3> a8 = std::move(a6); // 이동
14 }
```

- `std::array`의 타입이 이동을 지원하는 타입(movable)인 경우와 지원하지 않는 타입인 경우에 따라 다르게 동작 한다.
- `std::array`의 타입이 이동을 지원하지 않는 `std::array<int, 3>` 같은 경우 `std::move`도 동일하게 복사를 수행한다.
- 이동을 지원하는 `std::array<string, 3>` 같은 경우 a6의 모든 요소가 a8으로 이동(move)되었기 때문에 값은 사라져 있는 상태가 된다. 하지만 `std::array` 자체는 이동이 되지 않기 때문에 **크기는 동일한 3을 갖는다.**

3.4.10 stack, queue

`stack`, `queue`, 그리고 컨테이너 어댑터에 대해 살펴보자

```

1 int main() {
2     std::stack<int> s;
3     std::queue<int> q;
4
5     // # 요소 넣기
6     s.push(10);
7     s.push(20);
8     q.push(10);
9     q.push(20);
10
11    // # 요소 얻기
12    std::cout << s.top(); // 20
13    std::cout << q.front(); // 10
14
15    // # 요소 제거
16    s.pop();
17    q.pop();
18 }
```

`std::stack`의 구현 원리에 대해 살펴보자.

```

1 template<typename T, typename C = std::deque<T>>
2 class stack{
3     C c;
4 public:
5     void push(const T& a) { c.push_back(a); }
6     void pop() { c.pop_back(a); }
7     T& top() { return c.back(); }
8 };
9
10 int main() {
11     std::stack<int> s;
12     s.push(10);
13     s.push(20);
14
15     std::stack<int, std::list<int>> s1; // C에 list를 사용하겠다
16     std::stack<int, std::vector<int>> s2; // C에 vector를 사용하겠다
17 }
```

- `std::stack`은 기본적으로 `deque`를 내부에서 사용한다. 따라서 `pop`, `push`, `top`을 호출하면 내부적으로 `deque`의 메소드를 사용하여 요소에 접근한다.
- 결국 시퀀스 컨테이너의 "멤버 함수 이름을 변경"하여 스택처럼 사용할 수 있도록 만드는 것이다.
- 이러한 디자인 패턴을 **Adaptor 디자인 패턴**이라고 하며 따라서 `std::stack`을 컨테이너 어댑터라고 한다.

STL이 제공하는 컨테이너 어댑터는 다음과 같다.

c++98	<code>std::stack</code> <code>std::queue</code> <code>std::priority_queue</code>
c++23	<code>std::flat_set</code> <code>std::flat_multiset</code> <code>std::flat_map</code> <code>std::flat_multimap</code>

```
1 int main(){
2 // 다음 중 잘못된 코드를 찾아보자
3 std::stack<int, std::list<int>> s1;
4 std::stack<int, std::deque<int>> s2;
5 std::stack<int, std::vector<int>> s3;
6
7 std::queue<int, std::list<int>> q1;
8 std::queue<int, std::deque<int>> q2;
9 std::queue<int, std::vector<int>> q3; // error! 하지만 바로 에러가 발생하진 않는다
10
11 q3.push(10); // ok
12 q3.pop(); // error! 벡터에는 pop_front가 없으므로 에러가 발생한다
13
14 }
```

- `std::stack`: `back`, `push_back`, `pop_back`와 같이 3개의 멤버 함수가 필요하다. `std::list`, `std::vector`, `std::deque` 모두 사용 가능하다
- `std::queue`: `back`, `front`, `push_back`, `pop_front`와 같이 4개의 멤버 함수가 필요하다. `std::vector`는 `pop_front`가 없으므로 사용 불가능하다

```
1 int main(){
2 std::vector v{1,2,3};
3
4 // #1 스택 생성
5 std::stack<int> s1;
6 std::stack<int, std::vector<int>> s2{v}; // v의 요소를 복사
7
8 std::println("{} , {}", s2.size(), s2.top()); // 3,3
9
10 // #2 요소 넣기
11 s2.push(5);
12 s2.push_range(v); // c++23
13 s2.emplace(9); // c++11
14
15 std::println("{} , {}", s2.size(), s2.top()); // 8,9
16
17 // #3 반복자가 없다
18 auto p1 = s2.begin(); // error!
19 for(auto e : s2) {} // error!
20
21 }
```

3.4.11 priority_queue

또 다른 컨테이너 어댑터인 우선순위 큐(priority queue)에 대해 알아보자

```
1 int main(){
2 std::priority_queue<int> pq;
3
4 pq.push(3);
5 pq.push(1);
```

```

6 pq.push(2);
7
8 std::println("{}", pq.top()); // 3
9
10 pq.pop();
11
12 std::println("{}", pq.top()); // 2
13
14 pq.pop();
15
16 std::println("{}", pq.top()); // 1
17 }
```

- **priority_queue**: 우선 순위가 가장 높은 요소가 먼저 나오는 자료구조. 기본 세팅은 가장 큰 값이 나온다.

```

1 template<typename T,
2 typename Container = std::vector<T>,
3 typename Compare = std::less<
4 typename Container::value_type>>
5 class priority_queue;
```

- T: 요소의 타입

- Container: 내부적으로 사용하는 컨테이너. **random access iterator 조건을 만족하는 반복자를 가진 컨테이너만 사용 가능**하다(`std::vector`, `std::deque`). 기본적으로 `std::vector`를 사용한다

- Compare: 우선 순위를 결정할 때 사용하는 이항 함수 객체. 디폴트는 `std::less`를 사용한다.

`std::priority_queue`가 우선순위를 결정하는 방법은 3번째 템플릿 인자로 전달된 "**이항 함수 객체(타입)**"을 사용하여 요소를 비교하는 방법이다.

```

1 int main(){
2 std::priority_queue<int> pq;
3
4 pq.push(-3);
5 pq.push(1);
6 pq.push(2);
7
8 std::println("{}", pq.top()); // 2
9 }
```

- 우선 순위 결정 방식을 변경하려면 3번째 템플릿 인자로 "**이항 함수 객체(타입)**"을 전달해야 한다.
첫번째로 표준에 있는 함수 객체를 사용하는 방법이 있다.

```

1 int main(){
2 std::priority_queue<int, std::vector<int>, std::greater<int>> pq;
3
4 pq.push(-3);
5 pq.push(1);
6 pq.push(2);
7
8 std::println("{}", pq.top()); // -3
9 }
```

두번째로 사용자가 직접 만든 함수 객체를 사용하는 방법이 있다.

```

1 struct AbsLess{
2     bool operator()(int a, int b) const {
3         return abs(a) < abs(b);
4     }
5 }
6
7 int main(){
8     std::priority_queue<int, std::vector<int>, AbsLess> pq;
9
10    pq.push(-3);
```

```
11 pq.push(1);
12 pq.push(2);
13
14 std::println("{}", pq.top()); // -3
15 }
```

3.4.12 underlying container

컨테이너 어댑터가 내부적으로 사용하는 시퀀스 컨테이너에 접근하는 방법에 대해 알아보자

```
1 int main(){
2     std::priority_queue<int> pq;
3
4     pq.push(3);
5     pq.push(1);
6     pq.push(2);
7     pq.push(5);
8     pq.push(4);
9
10    std::println("{}", pq.top()); // 5.
11 }
```

위 코드에서 pq는 push할 때 정렬이 되는 걸까 아니면 top할 때 정렬이 되는 걸까?

```
1 template<typename T,
2 typename Container = std::vector<T>,
3 typename Compare = std::less<
4 typename Container::value_type>>
5 class priority_queue {
6 protected:
7     Container c;
8 public:
9 //...
10 }
```

- stack, queue, priority queue에 요소를 삽입하면 내부적으로 사용하는 시퀀스 컨테이너에 보관한다.
- 하지만 Container가 **protected**되어 있어서 외부에서 접근할 수 없다.
- priority queue를 상속하는 PrintableAdapter 클래스를 만들어 Container 멤버에 접근할 수 있다.

```
1 class PrintableAdapter : public std::priority_queue<int> {
2 public:
3     void print() const { show(this->c); }
4 };
5
6 int main(){
7     std::priority_queue<int> pq;
8
9     pq.push(3);
10    pq.push(1);
11    pq.push(2);
12    pq.push(5);
13    pq.push(4);
14
15    static_cast<PrintableAdapter&>(pq).print(); // 5,4,2,1,3
16 }
```

- **static_cast**<PrintableAdapter&>(pq) 같은 문법을 **다운캐스팅**이라고 한다. 다운캐스팅으로 멤버 변수에 접근할 수는 없으나 멤버 함수에는 접근할 수 있다.
- print() 결과 push하는 순간에 정렬되는 것을 알 수 있다.

priority queue의 내부 동작을 자세히 알기 위해 다음과 같은 코드를 작성해보자

```
1 struct log_less { // push될 때마다 실행되는 비교 연산자
```

```

2  bool operator()(int a, int b) const {
3      std::println("log less: {} {}", a,b,);
4      return a<b;
5  }
6  };
7
8  template<typename Adaptor>
9  struct PrintableAdaptor : public Adaptor {
10 void print() const { show(this->c); }
11 };
12
13 template<typename Adaptor>
14 void push_and_log(Adaptor& pq, int n) {
15     std::println("===== push ( {}) =====", n);
16     pq.push(n);
17     static_cast<PrintableAdapter<Adaptor>&>(pq).print();
18 }
19
20 template<typename Adapter>
21 void pop_and_log(Adapter& pq, int n) {
22     std::println("===== pop ( {}) =====", n);
23     pq.pop(n);
24     static_cast<PrintableAdapter<Adaptor>&>(pq).print();
25 }
26
27 int main() {
28     std::priority_queue<int, std::vector<int>, log_less> pq;
29
30     push_and_log(pq, 1); // push할 때마다 출력하라
31     push_and_log(pq, 3);
32     push_and_log(pq, 2);
33     push_and_log(pq, 4);
34
35     pop_and_log(pq); // pop할 때마다 출력하라
36     pop_and_log(pq);
37     pop_and_log(pq);
38     pop_and_log(pq);
39 }

```

- 출력 결과를 보면 less 비교 연산이 push, pop 될 때 모두 사용되는 것을 알 수 있다

3.4.13 std::set

- **시퀀스 컨테이너(sequence container)**: 요소들이 삽입된 순서를 유지하면서 한 줄로 놓여 있는 컨테이너 (`vector`, `deque`, `list`, `forwrad_list`, `array`)
- 시퀀스 컨테이너에 100개의 요소가 있을 때 값을 검색하려면 평균 50번의 이동과 비교가 필요하다. 좀 더 빠르게 검색할 수는 없을까?
- `tree`, `hash` 등 검색에 최적화된 자료구조를 사용하면 빠르게 검색 가능하며 이를 **연관 컨테이너(associative container)**라고 한다.
- associative라는 말은 `key` 값을 가지고 `value`를 보관하기 때문에 붙여진 이름이다

associative container 종류		
<code>set</code> <code>multiset</code>	<code>tree(red-black)</code>	key만 보관
<code>map</code> <code>multimap</code>	<code>tree(red-black)</code>	key-value 쌍 보관
<code>unordered_set</code> <code>unordered_multiset</code>	<code>hash</code>	key만 보관
<code>unordered_map</code> <code>unordered_multimap</code>	<code>hash</code>	key-value 쌍 보관

```

1 int main() {
2     std::set s{50,30,70,40,60};
3
4     // #1 요소 삽입 방법
5     s.insert(20);
6     s.emplace(80);
7     s.insert_range(std::vector{45, 55}); // c++23
8
9     // #2 중복을 허용하지 않음
10    auto ret = s.insert(40);
11
12    if(ret.second == false) { std::println("insert fail"); }
13 }
```

- `std::set`: key-value 쌍이 아닌 **key만 보관**한다. 대부분 red-black tree를 사용하여 구현되어 있다.
- `s.insert`의 반환값은 `std::pair<std::set<int>::iterator, bool>` `ret`이다. 두번째 인자를 검사하면 값이 삽입되었는지 유무를 알 수 있다.

3.4.14 find in `std::set`

`std::set`에서 요소를 검색하는 방법에 대해 알아보자

```

1 int main() {
2     std::set s{50,30,70,40,60,20,80};
3
4     auto ret1 = std::ranges::find(s, 70);
5
6     auto ret2 = s.find(70);
7
8     std::println("{}", *ret1);
9     std::println("{}", *ret2);
10 }
```

- 1) 알고리즘 `find` 사용: 1번째 요소(`s.begin()`)부터 `++` 연산자를 이용하여 검색. **동일(equality)** 요소 검색
- 2) 멤버함수 `find` 사용: `root`부터 크기를 비교하여 검색. **동등(equivalent)** 요소 검색

그래프로 나타내면 다음과 같다

```

1 50 (root)
2   \
3   30   70
4   / \   / \
5 20 40 60 80
6 (s.begin())
```

- `std::ranges::find(s, 70):` `s.begin()`부터 `++` 연산자를 사용해서 이동. `==` 연산자를 사용하여 동일 요소 검색 (**bad**)
- `s.find(70):` `root`부터 `(!((a<b) || (b>a)))` 연산으로 동등 요소 검색 (**good**)

동일성(equality)과 동등성(equivalent)의 차이점은 무엇일까?

```

1 struct MyCompare {
2     bool operator()(int a, int b) const {
3         return (a/10) < (b/10);      // 10의 자리수만 비교
4     }
5 };
6
7 int main() {
8     std::set<int, MyCompare> s;
9     s.insert(21);
10    s.insert(15);
11    s.insert(32);
12 }
```

```

13 auto ret = s.insert(17);
14 std::println("{}", ret.second); // false. MyCompare가 10의 자리수만 비교하기 때문에
15
16 auto ret1 = std::ranges::find(s, 12);
17 std::println("{}", ret1 == s.end()); // true. == 연산자로 탐색하기 때문에 12가 없으므로 true가
18 나온다
19
20 auto ret2 = s.find(12);
21 std::println("{} {}", ret2 == s.end(), *ret2); // false, 15 : 12가 없어도 MyCompare가 10의 자리수만
비교하기 때문에 15가 나온다
}

```

- `std::set`에 요소를 삽입할 때 `!(f(a,b) or f(b,a))`가 참이면 요소가 있다고 판단. 즉, 넣으려는 값보다 **크지도 않고 작지도 않으면** 요소가 있다고 판단한다

`set::contains` 멤버함수에 대해 알아보자. 이는 `set` 안에 특정 요소가 있는지를 조사할 때 사용한다.

```

1 int main() {
2     std::set s{50,30,70,40,60,20,80};
3
4     if(s.contains(70)) { std::println("s contains 70"); }
5
6     auto ret = s.find(70);
7
8     if(ret != s.end()) { std::println("s contains 70"); }
9 }

```

- `set::contains (c++20)`: `find`로 값을 찾고 `ret != s.end()`로 탐색 여부를 확인하는 두 번의 과정을 하나로 만들었다.

3.4.15 `std::set` with user defined type

`std::set`에 사용자 정의 타입(user defined type)을 보관하는 방법에 대해 살펴보자.

```

1 struct People {
2     int id;
3     std::string name;
4 };
5
6 int main() {
7     std::set<People> s;
8
9     s.emplace(1, "kim"); // error!
10    s.emplace(2, "lee");
11    s.emplace(3, "park");
12 }

```

- `People` 타입을 `std::set`에 보관하려면 2개의 `People` 객체에 대한 **비교 연산(<)**이 가능해야 한다.
- 비교 연산을 제공하는 3가지 방법
- 1) `set`의 템플릿 인자로 2개의 `People` 객체를 비교할 수 있는 함수 객체 전달
- 2) `People` 자체에 `operator()` 연산자 함수 제공
- 3) `std::less<People>`를 specialization

1) `People` 객체 2개를 비교할 수 있는 함수 객체 클래스(`PeopleCompare`)를 만든 후 `std::set`의 2번째 템플릿 인자로 전달하는 방법.

```

1 struct People {
2     int id;
3     std::string name;
4 };
5
6 struct PeopleCompare {
7     bool operator()(const People& p1,
8                      const People& p2) const {

```

```

9     return p1.id < p2.id;
10 }
11 };
12
13 int main() {
14     std::set<People, PeopleCompare> s;
15
16     s.emplace(1, "kim");
17     s.emplace(2, "lee");
18     s.emplace(3, "park");
19 }
```

2) People 타입 자체가 비교 연산을 제공하는 방법

```

1 struct People {
2     int id;
3     std::string name;
4
5 // c++98
6     bool operator<(const People& other) const {
7         return id < other.id;
8     }
9
10 // c++20
11     auto operator<=(const People& other) const = default;
12 };
13
14 int main() {
15     std::set<People> s; // std::set<People, std::less<People>> s; 와 동일
16
17     s.emplace(1, "kim");
18     s.emplace(2, "lee");
19     s.emplace(3, "park");
20 }
```

3) `std::less<People>`의 specialization 버전을 제공하는 방법

앞서 두번째 방법은 `std::less`를 사용하여 두 People 객체를 비교하는 방법이었다.

```

1 template<typename T = void>
2 struct less {
3     bool operator()(const T& a, const T& b) const {
4         return a < b;
5     }
6 };
```

세번째 방법은 위 `less`에서 특별한 버전을 새로 작성하는 것이다

```

1 struct People {
2     int id;
3     std::string name;
4 };
5
6 template<>
7 struct less<People> {
8     bool operator(const People& a, const People& b) const {
9         return p1.id < p2.id;
10    }
11 };
12
13 int main() {
14     std::set<People> s; // std::set<People, std::less<People>> s; 와 동일
15
16     s.emplace(1, "kim");
17     s.emplace(2, "lee");
18     s.emplace(3, "park");
19 }
```

-
- 개발자가 `std namespace` 안에 새로운 요소를 추가하는 것은 금지되어 있으나 `std` 안에 있는 템플릿을 specialization하는 것은 허용한다

3.4.16 `std::less` implementation

`std::less`, `std::ranges::less` 객체를 직접 구현하는 방법에 대해 알아보자

```
1 template<typename T>
2 struct less{                      // c++98
3     typedef T first_argument_type;
4     typedef T second_argument_type;
5     typedef bool result_type;
6
7     bool operator()(const T& a, const T& b) const {
8         return a < b;
9     }
10 };
11
12 int main(){
13     less<int> f1;
14
15     std::println("{}", f1(3, 4));    // true
16     std::println("{}", f1(3, 3.4)); // false (error)
17 }
```

- `typedef T ...` 부분은 c++98 시절에 사용되던 기술로 c++17에서 deprecated되고 c++20부터 제거되었다
- 위 코드의 문제점은 a와 b의 타입이 T이므로 `f1(3, 3.4)` 같은 코드를 처리하지 못한다

c++14부터 `less` 코드가 발전되었다.

```
1 template<typename T = void>
2 struct less{
3     bool operator()(const T& a, const T& b) const {
4         return a < b;
5     }
6 };
7
8 template<>
9 struct less{
10     template<typename T1, typename T2>
11     constexpr auto operator()(T1&& a, T2&& b) const
12     -> decltype(std::forward<T1>(a) < std::forward<T2>(b))
13     {
14         return std::forward<T1>(a) < std::forward<T2>(b);
15     }
16 };
17
18 int main(){
19     less<int> f1;      // c++98 less 사용
20     less<double> f1;
21
22     less<void> f3;    // c++14 이후 less 사용
23     less<> f3;
24     less f5;
25
26     std::println("{}", f1(3, 3.4)); // false;
27     std::println("{}", f5(3, 3.4)); // false;
28 }
```

- `T1 a, T2 b` 두 인자가 서로 다른 타입이다
- 완벽한 전달(perfect forwarding) 기술로 인자를 받고 있다
- 새롭게 작성한 `less` 코드는 c++98 `less`와 이름이 충돌하기 때문에 `less<void>`의 specialization 형태로 제공한다

다음으로 c++20 버전에서 추가되는 `less`에 대해 살펴보자

```

1  namespace std {
2
3      // c++98
4      template<typename T = void>
5      struct less{
6          bool operator()(const T& a, const T& b) const {
7              return a < b;
8          }
9      };
10
11     // c++14
12     template<>
13     struct less{
14         template<typename T1, typename T2>
15         constexpr auto operator()(T1&& a, T2&& b) const
16             -> decltype(std::forward<T1>(a) < std::forward<T2>(b))
17         {
18             return std::forward<T1>(a) < std::forward<T2>(b);
19         }
20
21         using is_transparent = int;
22     };
23
24     // c++20
25     namespace ranges {
26         struct less{
27             template<typename T1, typename T2> requires std::totally_ordered_with<T1, T2>
28             constexpr auto operator()(T1&& a, T2&& b) const
29                 -> decltype(std::forward<T1>(a) < std::forward<T2>(b))
30             {
31                 return std::forward<T1>(a) < std::forward<T2>(b);
32             }
33
34             using is_transparent = int;
35         };
36
37     }
38 }
39
40 int main(){
41     std::less<int> f1;
42     std::less f2;
43
44     std::ranges::less<int> f3; // error! 템플릿이 아님
45     std::ranges::less f4;      // ok
46 }
```

- c++20에서 추가된 `std::ranges::less`는 템플릿이 아니다.

3.4.17 is_transparent

함수 객체를 만들 때 자주 사용되는 `is_transparent`에 대해 살펴보자

```

1  struct People{
2      int id;
3      std::string name;
4  };
5
6  struct PeopleCompare {
7      bool operator()(const People& p1, const People& p2) const {
8          return p1.id < p2.id;
9      }
10 }
```

```

12 int main() {
13     std::set<People, PeopleCompare> s;
14     s.emplace(1, "lee");
15     s.emplace(2, "kim");
16
17     People p{1, "lee"};
18     auto ret = s.find(p);
19     std::println("{} {}", ret->name, ret->id);
20 }
```

- `std::find` 멤버 함수는 == 연산자가 아닌 템플릿 두번째 인자로 전달된 함수 객체를 사용하여 동등성(equivalence)를 조사한다. 즉, 지도 않고 작지도 않은 요소를 검사한다
- `PeopleCompare`는 `People`의 데이터 중 오직 id 값만을 사용하여 비교한다

`PeopleCompare`가 id만 검색하기 때문에 `auto ret = s.find(1)`과 같이 검색할 수는 없을까? `is_transparent`가 있으면 가능하다

```

1 struct People{
2     int id;
3     std::string name;
4 };
5
6 struct PeopleCompare {
7     bool operator()(const People& p1, const People& p2) const {
8         return p1.id < p2.id;
9     }
10
11    bool operator()(const People& p, int id) const {
12        return p.id < id;
13    }
14
15    bool operator()(int id, const People& p) const {
16        return id < p.id;
17    }
18
19    using is_transparent = int; // 아무 타입으로 선언해도 관계없음
20 };
21
22 int main() {
23     std::set<People, PeopleCompare> s;
24     s.emplace(1, "lee");
25     s.emplace(2, "kim");
26
27     People p{1, "lee"};
28     auto ret = s.find(p);
29     std::println("{} {}", ret->name, ret->id);
30 }
```

- `People` 객체와 `int`도 비교할 수 있도록 해야한다. 이를 위해서 비교 객체 안에 `is_transparent`가 반드시 선언되어 있어야 한다.

```

1 template<typename Key,
2 typename Pred = std::less<Key>,
3 typename Alloc = std::allocator<Key>>
4 class set {
5 public:
6     // find 인자로 set이 저장하는 타입(People)만 전달 가능
7     iterator find(const Key& key) {
8         Pred f;
9         f(root, key);
10    }
11
12    // find 자체가 함수 템플릿, 인자로 어떠한 타입도 받을 수 있다. 단, 비교 객체 함수 안에 is_transparent가 있을
13    // 때만 사용 가능하다 (SFINAE 기술)
14    template<typename Other,
```

```

14 typename = Pred::is_transparent>
15 iterator find(const Other& key) {
16     Pred f;
17     f(root, key);
18 }
19 };

```

People 자체에 연산자를 정의한 경우를 살펴보자

```

1 struct People{
2     int id;
3     std::string name;
4 };
5     bool operator<(const People& p1, const People& p2) { return p1.id < p2.id; }
6     bool operator<(const People& p, int id) { return p.id < id; }
7     bool operator<(int id, const People& p) { return id < p.id; }
8
9     int main() {
10         std::set<People> s;           // error! std::set<People, std::less<People>> s; 와 동일
11         std::set<People, std::less<void>> s; // ok
12
13         s.emplace(1, "lee");
14         s.emplace(2, "kim");
15
16         auto ret1 = s.find(People{1, "lee"}); // ok
17         auto ret2 = s.find(1);             // error!
18
19         std::println("{} {}", ret2->name, ret2->id);
20     }

```

- `std::less<People>`: (c++98) `is_transparent`가 없기 때문에 `s.find()` 템플릿 버전이 제공 안됨
- `std::less<void>`: (c++14) `is_transparent`가 존재하기 때문에 2개의 다른 타입 인자를 사용할 수 있음

3.4.18 `std::map`

<code>set</code>	tree(red-black)	key만 보관
<code>map</code>	tree(red-black)	key-value 짝 보관

- `std::map`의 요소는 `std::pair`이다
- 요소의 삽입은 pair를 만들어서 추가하는 것
- **반복자가 가리키는 요소는 `std::pair`이다.** `key`는 `iterator->first`로 접근하고 `value`는 `iterator->second`으로 접근한다

```

1 int main() {
2     std::map<std::string, std::string> m = {
3         {"mon", "Mon"}, {"tue", "Tue"}
4     };
5
6     // #1
7     m.insert(std::make_pair("wed", "Wed"));
8     m.insert({"thu", "Thu"});
9
10    // #2
11    m.emplace("fri", "Fri");
12
13    // #3
14    m["sat"] = "Sat";
15 }

```

- `std::map`는 `insert()`, `emplace()`, `[]` 연산자를 통해 요소를 추가할 수 있으며 이 중에서 `emplace()` 연산자를 권장한다

```
1 int main() {
2     std::map<std::string, std::string> m;
3     m.emplace("wed", "Wed");
4     m.emplace("thu", "Thu");
5     m.emplace("fri", "Fri");
6
7     auto first = m.begin();
8     auto last = m.end();
9
10    while( first != last ) {
11        std::cout << first->first << ", " << first->second << endl;
12        ++first;
13    }
14
15    // e는 std::pair
16    for(const auto& e : m)
17        std::cout << e.first << ", " << e.second << endl;
18
19    // c++17 structure binding 문법
20    for(const auto& [key, value] : m) {
21        std::cout << key << ", " << value << endl;
22    }
23 }
```

- `std::map`의 요소는 `std::pair`이기 때문에 `first`, `second`를 통해 `key`, `value`에 접근할 수 있다

```
1 int main() {
2     std::map<std::string, std::string> m;
3     m.emplace("sun", "Sun");
4
5     auto ret1 = m["sun"];           // "일요일"
6     auto ret2 = m["tue"];          // "", 예외가 발생하는게 아니라 요소를 추가하는 코드임에 유의!
7
8     for(const auto& [key,value] : m) {
9         std::cout << key << ", " << value << endl; // {"sun", "Sun"}, {"tue", ""}
10    }
11 }
```

- `std::map`의 `operator[]`
- `key`가 존재하는 경우 `value`를 반환
- `key`가 존재하지 않는 경우 `key`를 `map`에 추가하고 `value`는 디폴트 생성자로 초기화. 그리고 `value`를 반환

`std::map`에 특정 키 값이 있는지 조사하려면 다음과 같이 해야한다

```
1 int main() {
2     std::map<std::string, std::string> m;
3     m.emplace("sun", "Sun");
4
5     // # bad 새로운 항목을 추가하는 코드
6     if( m["mon"] != "" ) {}
7
8     // # good c++20 ~
9     if( m.contains("tue") ) {}
10
11    // # good ~ c++17
12    if( m.find("wed") != m.end() ) {}
13 }
```

3.5 Ranges

3.5.1 Range library

range-for 문을 사용해사 컨테이너의 "전체가 아닌 일부 요소만 출력"하고 싶을 경우 어떻게 해야할까?

```
1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     std::ranges::take_view tv{v, 5};           // 1,2,3,4,5 : v를 처음부터 5개까지 접근할 수 있는 뷰를 제공한다
5     std::ranges::reverse_view rv{v};          // 10,9,...,2,1 : v를 거꾸로 보는 뷰를 제공한다
6     std::ranges::reverse_view rv{tv};         // 5,4,3,2,1 : tv를 거꾸로 보는 뷰를 제공한다
7
8     for(auto e : tv) {
9         std::print("{} ", e);
10    }
11 }
```

- `std::ranges::take_view(c++20)`을 사용하면 된다
- range-for문의 원리는 반복자의 `begin()`, `end()`를 호출하여 모든 요소에 순차적으로 접근하는 방식이다

```
1 template<typename R>
2 class take_view {
3     R* rg;                                // 실제 코드는 std::ranges::ref_view<R> 사용
4     std::size_t const;
5 public:
6     take_view(R& r, std::size_t c) : rg(&r), count{c} {}
7
8     auto begin() { return rg->begin(); }
9     auto end() { return std::next(rg->begin(), count); }
10 }
```

- `end()`가 호출될 때 사용자가 임의로 정한 범위 `c`까지만 반환한다
- 자원을 소유하는 것이 아니라 자원을 소유한 `v`에 대한 **시각(view)**를 제공하는 것
- 따라서 `v`가 바뀌면 `take_view`도 같이 바뀐다

Cotainer	c++98 시절부터 사용되던 용어. 일반적으로 자원을 소유한다
Range	c++20에서 추가된 용어. <code>std::ranges::begin()</code> , <code>end()</code> 로 반복자를 얻을 수 있는 것
Borrowed range	자원을 소유하지 않은 Range. 다른 Range의 자원을 빌려서 사용 <code>std::ranges::string_view</code> <code>std::ranges::take_view</code> <code>std::ranges::reverse_view</code>

```
1 template<typename R>
2 void check(R& r) {
3     bool b1 = std::ranges::range<R>;
4     bool b2 = std::ranges::borrowed_range<R>;
5     std::println("{} {}", b1, b2);
6 }
7
8 int main() {
9     std::vector v{1,2,3,4,5,6,7,8,9,10};
10    std::ranges::take_view tv{v, 5};
11    std::ranges::reverse_view rv{v};
12
13    check(v);    // true, false
14    check(tv);   // true, true
15    check(rv);   // true, true
16 }
```

3.5.2 using views

view를 사용하는 다양한 예제를 살펴보자

```
1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     std::ranges::take_view tv{v, 5};           // 1,2,3,4,5
5     std::ranges::reverse_view rv{tv};          // 5,4,3,2,1
6     std::ranges::filter_view fv{rv,
7         [](int n) { return n%2==0; }}; // 4,2 : view는 중첩이 가능하다
8 }
```

- views: c++23 기준 30여개 이상 제공되고 있다
- view는 중첩이 가능하다

```
1 int main() {
2     std::vector v{1,2,3,4,5,6,7,8,9,10};
3
4     std::ranges::take_view tv{v, 5};
5
6     auto tv2 = std::views::take(v, 5);
7
8     auto tv3 = v | std::views::take(5);      // operator|() 연산자 재정의
9 }
```

- view를 만드는 세가지 방법
- 1) view 클래스 이름을 직접 사용하는 방법: take_view tv{v,5};
- 2) view를 만드는 함수를 사용하는 방법: auto tv = std::views::take(v,5);
- 3) 파이프 라인(|)을 통해 만드는 방법: auto tv = v | std::views::take(5);

파이프라인(|)을 사용하면 앞서 filter_view도 다음과 같이 작성할 수 있다

```
1 int main() {
2     auto fv = v | std::views::take(5) | std::views::reverse
3     | std::views::filter([](int n) { return n%2 == 0; });
4
5     for(auto e : v | std::views::take(5) ) { // range-for 문에도 파이프라인을 적용할 수 있다
6         std::print("{} ", e);
7     }
8 }
```

3.6 Utility

3.6.1 std::bind

std::bind 개념에 대해 알아보자

```
1 void foo(int a, int b, int c, int d) {
2     std::println("{} {} {} {}", a,b,c,d);
3 }
4
5 int main() {
6     foo(1,2,3,4);
7
8     auto f1 = std::bind(&foo, 1,2,3,4);    // foo의 4개 인자를 고정한 f1 함수 객체 생성
9     f1();
10
11    auto f2 = std::bind(&foo, 9,_1,8,_2); // foo에서 2,4번째 인자를 고정하지 않은 f2 함수 객체 생성
12    f2(5,7);
13 }
```

- std::bind: call wrapper라고도 불리며 callable object의 일부(전체) 인자를 고정한 새로운 함수 객체를 생성할 때 사용됨

- c++11에서 표준에 추가되었으며 <functional> 헤더가 필요함

```
1 void foo(int a, int b, int c, int d) {
2     std::println("{} {} {} {}", a,b,c,d);
3 }
4
5 int main() {
6     using namespace std::placeholders;
7
8     auto f = std::bind(&foo, 9,_2,8,_1);
9     f(5,7); // foo(9,7,8,5);
10 }
```

- _1, _2, _3: std::placeholders 안에 있는 미리 만들어져 있는 객체

- std::bind의 첫번째 인자는 함수의 reference가 들어가며 두번째부터 함수의 인자 개수에 따라 다음과 같은 4가지 종류의 값이 들어갈 수 있다

- 1) ordinary argument : 고정될 인자 값(9,8)
- 2) placeholders(1_, 2_)
- 3) std::reference_wrapper (using std::ref)
- 4) bind expression

```
1 void foo(int a) {
2     std::println("{}", a);
3 }
4
5 int main() {
6     int n = 0;
7
8     auto f1 = std::bind(&foo, n);
9     auto f2 = std::bind(&foo, std::ref(n));
10
11     n = 10;
12
13     f1(); // foo(0) : n 자체를 보관하는게 아니라 n이 가지고 있던 0의 값을 보관한다
14     f2(); // foo(10) : n을 참조로 보관하고 있기 때문에 n의 값이 바뀌면 호출도 바뀐다
15 }
```

- std::ref : reference wrapper를 생성하는 편의 함수 템플릿

유사한 다른 예제를 살펴보자

```
1 void foo(int a) { a = 100; }
2
3 int main() {
4     int n = 0;
5
6     auto f1 = std::bind(&foo, n);
7     auto f2 = std::bind(&foo, std::ref(n));
8
9     f1(); // n = 0
10    f2(); // n = 100 : n이 참조로 넘어가기 때문에 값이 변경된다
11 }
```

클래스 내에 멤버 함수를 bind하는 방법에 대해 알아보자

```
1 class Window{
2 public:
3     void move(int x, int y) {
4         std::println("{} {} {}", static_cast<void*>(this), x, y);
5     }
6 };
7
8 int main() {
```

```

9 Window w;
10 std::println("{}", static_cast<void*>(&w));
11
12 auto f1 = std::bind(&Window::move, w, 1, 2);           // 두번째 인자에 w를 입력하면 w의 복사본이 들어간다
13 auto f2 = std::bind(&Window::move, &w, 1, 2);         // &w와 같이 참조 형태로 전달해줘야 w 내의 move를
14                                         사용할 수 있다
15 auto f3 = std::bind(&Window::move, std::ref(w), 1, 2); // std::ref(w)도 &w와 동일
16
17 f1();
18 f2();
19 f3();
}

```

std::bind의 반환 타입을 알아보자

```

1 template<class F, class... Args>
2 constexpr /* unspecified */
3 bind(F&& f, Args&&... args);

```

- unspecified는 std::bind에 전달하는 인자에 따라 다른 타입이 반환된다는 것을 의미한다. 이를 C++ 표준에서는 명확하게 정의하지 않음(unspecified)을 의미한다

```

1 void foo(int a, int b, int c, int d) {
2   std::println("{} {} {} {}", a,b,c,d);
3 }
4
5 int main() {
6   auto f1 = std::bind(&foo, 1,2,3,4);
7   auto f2 = std::bind(&foo, 9,-1,8,-2);
8
9   std::println("{}", typeid(f1).name());
10  std::println("{}", typeid(f2).name());
11
12 f1.operator()();           // f1()
13 f2.operator()(5,6);        // f2(5,6)
14 }

```

std::bind는 멤버 함수 뿐만 아니라 멤버 변수도 bind할 수 있다.

```

1 struct Point {
2   int x,y;
3 };
4
5 int main() {
6   Point pt{1,2};
7
8   auto b = std::bind(&Point::y, &pt); // 멤버 변수도 bind 가능
9
10 std::println("{}", b());           // pt.y(2)
11
12 b() = 10;                      // pt.y = 10;
13
14 std::println("{}", pt.y());       // pt.y(10)
15 }

```

- std::bind는 callable한 것은 모두 bind할 수 있다

- callable하다는 것의 의미는 std::invoke()를 통해 호출 가능한 것들을 의미한다.

- 함수, 함수 포인터, 멤버 함수, 멤버 함수 포인터, 멤버 변수 포인터, 함수 객체, 람다 표현식 등등...

다음으로 bind expression이라는 개념에 대해 살펴보자

```

1 void foo(int a, int b, int c){
2   std::println("{} {} {}", a,b,c);
3 }
4

```

```

5 int main() {
6     int n = 0;
7
8     // 두번째 파라미터 n은 값으로 고정되고
9     // 세번째 파라미터 std::ref(n)은 참조값으로 고정된다.
10    // 네번째 파라미터 _1은 placeholders로 고정된다
11    auto f1 = std::bind(&foo, n, std::ref(n), _1);
12
13    n = 3;
14
15    f1(9);    // f1(0, 3, 9)
16}

```

```

1 int add(int a, int b) { return a+b; }
2 int mul(int a, int b) { return a*b; }
3
4 int main() {
5     auto f1 = std::bind(mul, _1, 2);
6
7     f1(10);           // mul(10, 2)
8
9     auto f2 = std::bind(&mul, std::bind(&add, _1, _2), 2);
10
11    f2(3,4);         // mul(add(3,4), 2)
12
13    auto f3 = std::bind(&add, _1, _2);
14    auto f4 = std::bind(&mul, f3, 2);
15
16    f4(3,4);         // mul(add(3,4), 2)
17}

```

- **bind expression** : `std::bind`의 인자로 다시 `std::bind`를 사용할 수 있다. 함수 결합(function composition) 기법이라고도 불린다

bind expression 없이 바로 `std::bind(&mul, &add, 2)`와 같은 형태를 사용할 수는 없을까?

```

1 int mul(int a, int b) { return a*b; }
2
3 struct Add {
4     int operator()(int a, int b) const { return a+b; }
5 };
6
7 template<>
8 struct std::is_bind_expression<Add> : public std::true_type{};
9
10 int main() {
11     std::println("{}", std::is_bind_expression_v<Add> );
12
13     auto f1 = std::bind(&mul, Add{}, 2);
14
15     std::println("{}", f1(3,4));
16 }

```

- `std::bind(&mul, ??, 2)`와 같이 두번째 인자에 다른 함수(callable type)를 사용하려면 `std::is_bind_expression_v<callable type>`이 나오도록 callable type을 만들어야 한다.

- `add`를 함수가 아닌 함수 객체로 설계하고 `std::is_bind_expression<Add>`에 대한 specialization을 제공해야 한다

c++20,23에서 추가된 `bind_first`, `bind_back`에 대해 살펴보자

```

1 void foo(int a, int b, int c, int d){
2     std::println("{} {} {} {}", a,b,c,d);
3 }
4
5 int main() {

```

```

6 int n = 0;
7 auto f1 = std::bind(&foo, 0, 0, _1, _2);
8 auto f2 = std::bind_front(&foo, 0, 0);
9
10 f1(1,2);
11 f2(1,2);
12
13 auto f3 = std::bind(&foo, _1, _2, _3, 0);
14 auto f4 = std::bind_back(&foo, 0);
15
16 f3(1,2);
17 f4(1,2);
18 }
```

- `std::bind_first`, `std::bind_back` : 인자를 앞쪽(또는 뒤쪽)부터 차례대로 고정할 때 사용하며 `placeholder`를 사용하지 않는다

- `placeholder`를 사용하지 않기 때문에 인자의 재정렬(rearrangement)이 지원되지 않는다

3.6.2 `std::function`

함수 포인터(function pointer)란 함수의 주소를 보관했다가 호출할 때 사용하는 포인터로써 callback 함수 개념에서 널리 사용한다. 함수 포인터의 단점은 다음과 같다

- `signature`가 동일한 함수의 주소만 보관할 수 있다.
- 멤버 함수와 비멤버 함수의 주소를 보관하고 호출하는 방법이 다르다

```

1 void f1(int a) {}
2 void f2(int a, int b) {}

3
4 struct Window {
5     void set_width(int w) {}
6 };

7
8 int main() {
9     void (*f)(int);

10    f = &f1;           // ok
11    f = &f2;           // error. 함수 포인터는 signature가 동일한 함수 주소만 보관할 수 있다
12    f = &Window::set_width; // error. 멤버 함수의 주소를 담을 수 없다
13
14
15    int n = 0;
16    f = [] (int a) {};      // ok
17    f = [n] (int a) {};    // error. 캡처한 람다 표현식은 함수 포인터로 암시적 변환될 수 없다
18 }
```

`std::function`은 일반화된 목적의 함수 래퍼(wrapper)이다. 이를 통해 함수 포인터의 단점을 보완할 수 있다.

```

1 void f1(int a) {}
2 void f2(int a, int b) {}

3
4 struct Window {
5     void set_width(int w) {}
6 };

7
8 int main() {
9     std::function<void(int)> f = &f1;
10
11    f(10);           // f1(10)
12
13    int n = 0;
14    f = [] (int a) {};      // ok
15    f = [n] (int a) {};    // ok. 캡처한 람다 표현식도 가능
16
17    f = &f2;           // error. 인자 2개짜리 함수 포인터는 담지 못함
18 }
```

```

18 f = std::bind(&f2, _1, 0);           // ok. std::bind를 사용하면 인자 1개짜리 함수 포인터로 사용 가능
19
20 Window w;
21 f = &Window::set_width;          // error. 멤버 함수는 불가능
22 f = std::bind(&Window::set_width, &w, _1); // ok
23
24 f(10);                         // w.set_width(10)
25

```

- `std::function<void(int)>` : `void`를 반환하며 인자가 `int` 한 개인 호출 가능한 모든 것(callable object)를 담겠다는 의미

```

1 void foo(int a) {}
2
3 int main() {
4     std::function<void(int)> f1{ &foo };
5     std::function<int(int, int)> f2{ std::plus<int>{} };
6     std::function<void(int)> f3{ [](int a) {} };
7
8     std::println("{}", f1.target_type().name());
9     std::println("{}", f2.target_type().name());
10    std::println("{}", f3.target_type().name());
11
12    auto ptr1 = f1.target<void(*)(int)>();
13    auto ptr2 = f1.target<int*>();
14
15    std::println("{}", *ptr1 == foo );
16    std::println("{}", ptr2 == nullptr );
17

```

- `std::function`의 callable target : `std::function` 객체가 내부적으로 보관하고 있는 호출 가능한 요소
- `target_type()` : target의 `typeid`를 반환. (`const std::type_info&` 타입)
- `target<>()` : target의 주소

```

1 int main() {
2     std::function<void(int)> f;
3
4     try {
5         f(10);
6     }
7     catch(const std::bad_function_call& e) {
8         std::println("exception : {}", e.what());
9     }
10
11    if( f ) std::println("has target"); // std::function은 bool로 명시적 변환 가능
12    else    std::println("no target");
13

```

- `std::bad_function_call` 예외 : target이 없는 `std::function` 객체에 대해 ()을 사용하여 호출하는 경우 발생하는 예외
- `std::function`은 `bool`로 명시적 변환이 가능하므로 `if`문으로 target이 있는지 조사할 수 있다

3.6.3 `std::reference_wrapper`

c++ 표준에 정의된 `std::reference_wrapper`, `std::ref`, `std:: cref`에 대해 살펴보자. 이에 앞서 우선 포인터와 참조의 차이점에 대해 알아보자

```

1 int main() {
2     int n = 10;
3
4     int* p = &n;    // 포인터
5     int& r = n;    // 참조

```

```
6
7 *p = 20;
8 r = 20;
9 }
```

-
- 포인터와 참조의 차이점을 먼저 알아보자
 - 포인터는 자신도 변경할 수 있고 대상체의 값도 변경할 수 있다
 - 참조도 내부적으로 포인터 연산을 사용하고 있지만 **자신을 변경하는 것이 불가능한** 차이점이 존재한다

```
1 int main() {
2     int n1 = 10;
3     int n2 = 20;
4
5     std::reference_wrapper r1(n1);
6     std::reference_wrapper r2(n2);
7
8     r1 = r2;           // p1 = p2; 포인터처럼 동작하여 r1이 n2를 가리키도록 변경된다
9     r1.get() = r2.get(); // *p1 = *p2; r1이 가리키는 대상체(n1)의 값을 변경한다
10
11    std::println("{} {}", n1, n2);
12    std::println("{} {}", r1.get(), r2.get());
13 }
```

-
- **std::reference_wrapper** : c++11에서 처음 등장하였으며 <functional> 헤더가 필요하다. 이동 가능한 참조라고도 불리며 대입 연산 시 포인터와 유사하게 동작하도록 설계되었다. 포인터의 wrapper 클래스라고도 불린다

std::reference_wrapper의 구현 원리에 대해 살펴보자

```
1 template<typename T> class reference_wrapper {
2     T* pobj{};
3
4     public:
5     reference_wrapper(T& obj) : pobj{&obj} {}
6
7     int main() {
8         int n1 = 10;
9         int n2 = 20;
10
11        reference_wrapper r1{n1};
12        reference_wrapper r2{n2};
13
14        r1 = r2;           // 컴파일러에 의해 기본적인 같은 복사가 일어남. 포인터와 동일하게 주소를 넘겨주는
15        형태로 동작
16    }
17 }
```

-
- 대입 연산자(=)를 정의해주지 않아도 같은 복사가 일어나서 포인터와 동일하게 주소를 넘겨준다

```
1 template<typename T> class reference_wrapper {
2     T* pobj{};
3
4     public:
5     reference_wrapper(T& obj) : pobj{&obj} {}
6
7     operator T&() const { return *pobj; }
8
9     T& get() const { return *pobj; }
10
11    int main() {
12        int n = 10;
13
14        reference_wrapper r1{n};
15
16        int &r2 = r1;           // 진짜 참조(&)로 암시적 변환이 가능하다
17        int &r3 = r1.get();    // 위와 동일
18 }
```

 }

- 핵심1 : `std::reference_wrapper`는 진짜 참조(&)로 암시적 변환이 가능하다

```

1 int add(int a, int b) { return a+b; }

2
3 int main() {
4     std::reference_wrapper f{add}; // 함수도 사용 가능
5
6     int n = f(1,2);           // f.operator()(1,2)
7
8     std::println("{}", n);
9 }
```

- 핵심2 : `std::reference_wrapper` 자체도 callable object이므로 `operator()` 연산자를 제공하며 `add` 함수의 참조도 만들 수 있음을 의미한다

`std::reference_wrapper`의 객체를 생성하는 방법에 대해 살펴보자

```

1 int main() {
2     int n = 10;
3
4     // 1) 클래스 템플릿 사용
5     std::reference_wrapper r1{n};
6     std::reference_wrapper<int> r2{n};
7     std::reference_wrapper<const int> r3{n};
8
9     r1.get() = 20;
10    r2.get() = 20;
11    r3.get() = 20; // error! const int&
12
13 // 2) 편의 함수 템플릿 사용
14 auto r4 = std::ref(n); // r1, r2와 동일
15 auto r5 = std:: cref(n); // r3와 동일
16
17 r4.get() = 20;
18 r5.get() = 20; // error! const int&
19 }
```

- 1) 클래스 템플릿을 직접 사용하는 방법

- 2) `std::ref()`, `std::cref()`과 같은 편의 함수 템플릿을 사용하는 방법

`std::reference_wrapper`를 활용하는 예제를 살펴보자

```

1 void foo(int& r) { r = 100; }

2
3 template<typename F, typename T>
4 void log_and_call(F f, T arg) {
5     // logging
6     f(arg);
7 }

8
9 int main() {
10     int n = 0;
11
12     log_and_call(foo, n);           // bad! T arg가 call-by-value로 받기 때문에 n이 변경되지 않는다.
13
14     log_and_call(foo, &n);         // bad! 포인터는 int& r에서 참조로 변환될 수 없으므로 컴파일 에러 발생
15
16     log_and_call(foo, std::ref(n)) // ok
17
18     std::println("{}", n);
19 }
```

-
- 값으로 설계된 라이브러리(코드)에 참조를 사용하고 싶을 때 주로 사용된다
 - 쉽게 말해서 **call-by-value로 설계된 코드에 call-by-reference를 보내고 싶을 때 사용**한다

`std::reference_wrapper`를 사용하지 말고 인자를 처음부터 참조로 받으면 되지 않을까?

- 기본 정책으로 call-by-reference보다 call-by-value를 사용하는 것이 안전한 경우가 많이 있다
- `std::bind`, `std::thread` 함수들이 대표적으로 call-by-value를 기본 정책으로 사용한다

```
1 void foo(int a) { a = 100; }
2
3 int main() {
4     std::function<void()> f;
5
6     {
7         int n = 0;
8         f = std::bind(foo, n); // n을 값으로 보관해야 스코프를 지나도 값이 파괴되지 않는다
9     }
10
11 f(); // foo(0)
12 }
```

- `std::bind`: 내부적으로 인자를 bind할 때 값(복사본)을 보관한다. 값이 아닌 참조로 bind하고 싶다면 `std::ref()`를 사용하면 된다

컨테이너와 `std::reference_wrapper`의 관계를 알아보자

```
1 struct Point {
2     int x,y;
3 };
4
5 int main() {
6     Point p1{1,1};
7     Point p2{1,1};
8
9     std::vector<Point&> v; // error! 컨테이너는 참조를 보관할 수 없다
10    std::vector<std::reference_wrapper<Point>> v; // ok
11
12    v.push_back(p1);
13    v.push_back(p2);
14
15    v[0].get().x = 20;
16
17    std::println("{}", p1.x); // 20
18 }
```

- 자주 사용되는 기법은 아니지만 이렇게 사용할 수도 있으니 알아놓으면 좋다

3.6.4 Smart pointer

스마트 포인터란 포인터와 유사하게 동작하는 추상화된 타입으로 포인터 기능 외에 **자동화된 자원 관리 기능**을 추가로 제공하는 포인터를 말한다.

```
1 class Car {
2     int color;
3     public:
4     ~Car() { cout << "~Car()" << endl; }
5     void Go { cout << "Car go" << endl; }
6 };
7
8 int main() {
9     Car* p = new Car;
10
11     p->Go();
```

```
12 (*p).Go();  
13  
14 delete p; // 메모리 누수 방지를 위해 포인터는 쓰고나면 반드시 제거해줘야 한다  
15 }
```

```
1 class Car {  
2     int color;  
3 public:  
4     ~Car() { cout << "~Car()" << endl; }  
5     void Go { cout << "Car go" << endl; }  
6 };  
7  
8 int main() {  
9     shared_ptr<Car> p( new Car );  
10  
11     p->Go(); // p.operator->() 호출  
12     (*p).Go(); // p.operator*() 호출  
13 }
```

- `shared_ptr<Car>`는 실제 포인터는 아니고 객체이므로 스코프를 벗어날 때 자동으로 소멸자가 불려 파괴된다

스마트 포인터 중 하나인 `shared_ptr<>`에 대해 살펴보자.

```
1 #include <memory>  
2  
3 class Car {  
4     int color;  
5     int speed;  
6 public:  
7     Car(int c = 0, int s = 0) : color(c), speed(s) {}  
8     ~Car() { cout << "~Car()" << endl; }  
9     void Go { cout << "Car go" << endl; }  
10 };  
11  
12 int main() {  
13     int a = 0; // 복사 초기화  
14     int a(0); // 직접 초기화  
15  
16     shared_ptr<Car> p = new Car; // error! 복사 초기화는 불가능하다  
17     shared_ptr<Car> p(new Car); // ok. 직접 초기화만 가능  
18  
19     shared_ptr<Car> p1(new Car);  
20     shared_ptr<Car> p2 = p1; // 제어 블록에서 use_count 값이 2가 된다  
21 }
```

- 스마트 포인터를 사용하려면 `<memory>` 헤더가 필요하다
- 스마트 포인터는 `explicit` 생성자이기 때문에 복사 초기화가 불가능하고 직접 초기화만 가능하다
- `shared_ptr<>`은 포인터 외에도 참조 계수 등을 관리하는 **제어 블록(control block)**을 가리키고 있으며 이 중 `use_count` 값이 현재 몇개의 포인터가 가리키고 있는가를 의미한다

`shared_ptr<>`의 삭제자(deleter)를 변경하는 방법에 대해 살펴보자

```
1 void foo(Car* p) {  
2     cout << "Delete Car" << endl;  
3     delete p;  
4 }  
5  
6 int main() {  
7     shared_ptr<Car> p( new Car, foo ); // 두번째 인자로 함수가 파괴될 때 실행되는 함수를 정의할 수 있다  
8  
9     shared_ptr<Car> p( new Car, [](Car* p) { // 람다 표현식도 사용할 수 있다  
10         cout << "Delete Car" << endl;  
11         delete p;  
12     } );
```

13 }

- 두번째 인자로 함수가 파괴될 때 실행되는 함수를 정의할 수 있다(삭제자)
- 제어 블록의 deleter에 함수가 등록되어 파괴될 때 실행된다

shared_ptr<>의 할당자(allocator)를 변경하는 방법에 대해 살펴보자

```
1 template<typename T>
2 class MyAlloc {
3 public:
4     using value_type = T;
5     MyAlloc() noexcept {}
6
7     template<typename U>
8     MyAlloc(const MyAlloc<U>&) noexcept {}
9
10    T* allocate(std::size_t num) {
11        std::cout << typeid(T).name() << std::endl;
12        std::cout << "size : " << num << std::endl;
13        return static_cast<T*>(::operator new(sizeof(T)*num));
14    }
15
16    void deallocate(T* p, std::size_t num) {
17        std::cout << "delete : " << num << std::endl;
18        ::operator delete(p);
19    }
20 ...
21 };
22
23 int main() {
24     shared_ptr<Car> p( new Car,
25     [](Car* p) { ... },
26     MyAlloc<Car>()); // 세번째 인자에 할당자(allocator)를 설정할 수 있다
27 }
```

- 제어 블록의 allocator가 할당자의 정보를 담고 있으며 할당자는 제어 블록을 만들고 파괴하는 방법이 정의되어 있다

shared_ptr<>에 배열을 담을 수 있을까?

```
1 int main() {
2     // c++14 - 128) * 64 + (' - 128) - 128) * 64 + (' - 128) 뼘
3     shared_ptr<Car> p1( new Car[10], [](Car* p) {delete[] p;} );
4
5     p1[0].Go(); // error!
6
7
8     // c++17 - 128) * 64 + (' - 128) 뼘 - 128) * 64 + (' - 128) 퀼
9     shared_ptr<Car[]> p1( new Car[10] );
10
11    p1[0].Go(); // ok
12 }
```

- c++14 까지 : shared_ptr<>에 배열을 넣으려면 삭제자를 배열 관련된 삭제자로 변경해줘야 한다
- shared_ptr<>은 [] 연산자가 제공되지 않으므로 배열과 같이 사용할 수 없다. std::vector, std::array를 통해 배열을 관리할 것을 권장한다
- c++17 이후 : shared_ptr<...[]> 과 같이 스마트 포인터도 배열을 지원하게 되었다

shared_ptr<>의 다양한 특징들에 대해 살펴보자

```
1 int main() {
2     shared_ptr<Car> p1( new Car );
3
4     p1->Go(); // Car의 멤버 접근
5 }
```

```

6 Car* p = p1.get();           // shared_ptr 자체 멤버에 접근
7
8 shared_ptr<Car> p2 = p1;
9 int n = p1.use_count();
10
11 p1.reset( new Car );
12
13 p1.swap(p2);
14

```

- -> 연산 : 대상 객체(Car)의 멤버에 접근
- . 연산 : shared_ptr 자체 멤버에 접근

get	대상체의 포인터 반환
use_count	참조계수 반환
reset	대상체 변경
swap	대상체 교환

make_shared 함수에 대해 살펴보자

```

1 void* operator new(size_t sz) {           // new 연산자 재정의
2 cout << "new sz : " << sz << endl;
3 return malloc(sz);
4 }

5
6 int main() {
7 shared_ptr<Car> p1( new Car );          // 객체(8 byte), 제어블록(24 byte) 출력
8
9 shared_ptr<Car> p1 = make_shared<Car>(); // 객체 + 제어 블록을 한 공간에 메모리 할당함
10 shared_ptr<Car> p1( make_shared<Car>() );
11

```

- shared_ptr<>를 생성하면 객체를 위한 메모리 할당과 제어 블록을 위한 메모리 할당이 별도로 발생한다. 이는 메모리가 조각화되어 속도가 느려지는 원인이 된다
- make_shared를 사용하여 초기화하면 **객체와 제어 블록의 메모리를 한 공간에 할당**할 수 있어 메모리를 효율적으로 사용할 수 있다

```

1 int main() {
2 f( shared_ptr<Car>(new Car), foo() ); // danger!
3
4 f( make_shared<Car>(), foo() );       // safe!
5
6 shared_ptr<Car> p1 = allocate_shared<Car>( MyAlloc<Car>() ); // allocate_shared을 사용하면 할당자를
7   설정할 수 있다

```

- 또한 make_shared는 예외 상황에 좀 더 안전하다
- new Car, shared_ptr, foo() 함수 호출 순서가 표준에 정해져 있지 않다. 만약 foo에서 예외가 발생하면 shared_ptr에서 메모리 해제 코드가 없기 때문에 메모리 누수가 발생한다
- allocate_shared는 make_shared와 유사하나 할당자를 추가할 수 있다는 점이 다르다
- 이와 같은 장점들 때문에 **shared_ptr을 사용할 때는 make_shared와 같이 사용하는 것이 매우 권장된다**

shared_ptr<>을 사용할 때 주의할 점에 대해 살펴보자

```

1 int main() {
2 Car* p = new Car;
3
4 shared_ptr<Car> sp1(p);           // 제어 블록 생성
5 shared_ptr<Car> sp2(p);           // 또 다른 제어 블록 생성
6
7
8 shared_ptr<Car> sp3( new Car )    // better : shared_ptr 만들 때 포인터를 선언한다(RAII)
9 shared_ptr<Car> sp4 = make_shared<Car>(); // best

```

10 }

- 포인터를 사용하여 2개 이상의 `shared_ptr`을 생성하면 안된다!
- 따라서 포인터를 만들고 `shared_ptr`을 생성하지 말고 `shared_ptr`을 생성할 때 포인터를 만드는 것이 권장된다 (**RAII(Resource Acquisition Is Initialization)**)

조금 어려운 개념인 `enable_shared_from_this`에 대해 살펴보자

```
1 class Worker {
2     Car c;
3     public:
4     void Run() {
5         thread t(&Worker::Main, this);
6         t.detach();
7     }
8
9     void Main() {
10     c.Go();      // 멤버 데이터(Car) 사용
11     cout << "finish thread" << endl;
12 }
13 };
14
15 int main() {
16 {
17     shared_ptr<Worker> sp = make_shared<Worker>();
18     sp->Run();
19 }
20 // 위 스코프를 지나면서 use_count가 0이 되어 Worker가 파괴된다
21 ...
22 }
```

- Worker 객체는 스코프가 지나도 살아있어야 한다
- 다시 말하면, 주 스레드의 `sp1`도 사용하지 않고 새로운 스레드도 종료되었을 때 파괴되어야 한다
- Worker 객체가 자신의 참조 계수(use_count)를 증가시켜야 한다

```
1 class Worker : public enable_shared_from_this<Worker> // CRTP
2 {
3     Car c;
4     shared_ptr<Worker> holdMe;
5
6     public:
7     void Run() {
8         holdMe = shared_from_this(); // 자기 참조 계수 1 증가시킴
9
10    thread t(&Worker::Main, this);
11    t.detach();
12 }
13
14 void Main() {
15     c.Go();      // 멤버 데이터(Car) 사용
16     cout << "finish thread" << endl;
17
18     holdMe.reset(); // 파괴
19 }
20 };
21
22 int main() {
23 {
24     shared_ptr<Worker> sp = make_shared<Worker>();
25     sp->Run();
26 }
27 ...
28 }
```

-
- enable_shared_from_this : `this`를 가지고 제어 블록을 공유하는 `shared_ptr`을 만들 수 있게 한다 (**CRTPO 기술**)
 - `shared_from_this()`를 호출하기 전에 반드시 제어 블록이 생성되어 있어야 한다

3.6.5 weak_ptr

스마트 포인터와 상호 참조 문제에 대해 알아보자

```
1 struct People {
2 People(string s) : name(s) {}
3 ~People() { cout << "~People : " << name << endl; }
4
5 string name;
6 shared_ptr<People> bf; // best friend
7 };
8
9 int main() {
10 shared_ptr<People> p1( new People("KIM"));
11 shared_ptr<People> p2( new People("LEE"));
12
13 p1->bf = p2;
14 p2->bf = p1; // 상호참조 버그 발생!
15 // 코드가 끝나고 포인터가 파괴되지 않는다
16 }
```

- `bf`로 서로를 가리킬 때는 참조 계수가 증가하면 안된다

위 코드는 상호 참조 문제가 존재한다. 이를 해결하기 위해 raw pointer를 사용할 수 있다

```
1 struct People {
2 People(string s) : name(s) {}
3 ~People() { cout << "~People : " << name << endl; }
4
5 string name;
6 People* bf; // Raw pointer 사용
7 };
8
9 int main() {
10 shared_ptr<People> p1( new People("KIM"));
11 shared_ptr<People> p2( new People("LEE"));
12
13 p1->bf = p2.get();
14 p2->bf = p1.get(); // ok
15 }
```

하지만 raw pointer는 아래와 같은 경우 문제가 발생한다

```
1 struct People {
2 People(string s) : name(s) {}
3 ~People() { cout << "~People : " << name << endl; }
4
5 string name;
6 People* bf;
7 };
8
9 int main() {
10 shared_ptr<People> p1( new People("KIM"));
11 {
12     shared_ptr<People> p2( new People("LEE"));
13
14     p1->bf = p2.get();
15     p2->bf = p1.get();
16 } // p2 자원 파괴
17
18 if(p1->bf != 0)
19     cout << p1->bf->name << endl; // error!
```

20 }

- p2는 파괴됐지만 p1->bf는 리셋되지 않기 때문에 에러가 발생한다
- raw pointer : 참조 계수가 증가하지 않지만 원본 객체가 파괴되었는지 알 수 없는 단점이 존재한다

참조 계수가 증가하지 않으면서 원본 객체가 파괴되었는지를 확인할 수 있는 방법은 없을까?

```
1 int main() {
2     weak_ptr<Car> wp;
3     shared_ptr<Car> sp(new Car);
4
5     wp = sp;
6
7     cout < sp.use_count() << endl; // 1
8 }
```

- weak_ptr<>은 use_count가 증가하지 않는다

```
1 int main() {
2     weak_ptr<Car> wp;
3
4     {
5         shared_ptr<Car> sp(new Car);
6         wp = sp;
7         cout < sp.use_count() << endl;
8     } // sp 파괴
9
10    if( wp.expired() ) { cout << "destroy" << endl; }
11    else { cout << "not destroy" << endl; }
12 }
```

- weak_ptr<> : use_count가 증가하지 않으면 expired() 함수로 대상 객체의 유효성을 판단할 수 있다

```
1 int main() {
2     weak_ptr<Car> wp;
3
4     shared_ptr<Car> sp(new Car);
5     wp = sp;
6     cout < sp.use_count() << endl;
7
8     if( wp.expired() ) { cout << "destroy" << endl; }
9     else {
10        cout << "not destroy" << endl;
11
12        wp->Go() // error!
13
14        shared_ptr<Car> sp2 = wp.lock();
15
16        if(sp2) { sp2->Go(); }
17    }
18 }
```

- weak_ptr<>은 대상 객체에 접근할 수 없다. 접근하기 위해서는 weak_ptr<>을 가지고 다시 shared_ptr<>을 만들어야 한다

앞선 예제를 weak_ptr<>로 다시 작성해보자

```
1 struct People {
2     People(string s) : name(s) {}
3     ~People() { cout << "~People : " << name << endl; }
4
5     string name;
6     weak_ptr<People> bf;
```

```

7 } ;
8
9 int main() {
10 shared_ptr<People> p1( new People("KIM"));
11 {
12     shared_ptr<People> p2( new People("LEE"));
13
14     p1->bf = p2.get();
15     p2->bf = p1.get();
16 } // p2 자원 파괴
17
18
19 if(p1->bf.expired()) {} // 유효성 확인
20
21 shared_ptr<People> sp2 = p1->bf.lock(); // 객체 접근
22
23 if(sp2) { cout << sp2->name << endl; }
24 else { cout << "destroy bf" << endl; }
25 }
```

`weak_ptr<>`의 원리에 대해 살펴보자

```

1 int main() {
2 weak_ptr<Car> wp;
3
4 {
5     shared_ptr<Car> sp( new Car );
6     wp = sp; // weak_count 1 증가
7 }
8
9 if( wp.expired() ) {}
10 }
```

- 제어 블록 내에 `weak_count`가 존재하여 `wp`가 `Car`를 가리키게 되면 `use_count`가 아닌 `weak_count=1`로 카운트된다
- 따라서 `weak_ptr<>`은 제어 블록 내에 `use_count=0 && weak_count==0`일 때 파괴된다

3.6.6 unique_ptr

`unique_ptr`이라는 스마트 포인터에 대해 살펴보자

```

1 int main() {
2 shared_ptr<Car> sp1( new Car );
3 shared_ptr<Car> sp2 = sp1; // use_count = 2
4
5 unique_ptr<Car> up1( new Car ); // 자원 독점
6 unique_ptr<Car> up2 = up1; // error
7
8
9 cout << sizeof(sp1); << endl; // 16
10 cout << sizeof(up1); << endl; // 8 (크기가 작다)
11 }
```

- `unique_ptr`은 자원을 공유하지 않고 메모리 관리를 자동으로 해주는 포인터를 원할 때 사용하면 된다
- 기본적으로 raw pointer와 동일한 크기를 가진다. 단, 삭제자(deleter) 변경 시 크기가 커질 수 있다

```

1 int main() {
2 unique_ptr<int> up1( new int );
3 unique_ptr<int> up2 = std::move(up1); // ok. 복사될 수 없지만 이동할 수는 있다
4 }
```

- `unique_ptr`은 복사할 수 없지만 이동할 수 있다

`unique_ptr`에 삭제자는 어떻게 적용할까?

```

1 void foo(int* p) {
2     cout << "foo" << endl;
3     delete p;
4 }
5
6 struct Deleter {
7     void operator()(int* p) const {
8         delete p;
9     }
10 }

11 int main() {
12     shared_ptr<int> sp( new int, foo );
13
14     // 1. 함수 객체 사용
15     unique_ptr<int, Deleter> up( new int );
16
17     // 2. 함수 포인터 사용
18     unique_ptr<int, void(*)(int*)> up( new int, foo );
19
20     // 3. 람다 표현식 사용
21     auto f = [](int* p) { delete p; }
22     unique_ptr<int, decltype(f)> up( new int, f );
23 }

```

- 삭제자를 적용하는 방법에는 위와 같은 3가지 방법이 있다

```

1 int main() {
2     unique_ptr<int[]> up( new int[10] ); // 배열 포인터 지정
3 }

```

- 배열을 사용하려면 타입 `int[]`과 배열 크기 `int[10]`을 동시에 명시해줘야 한다

- `shared_ptr`은 C++17부터 배열에 대한 포인터를 지원하지만 `unique_ptr`은 C++11부터 지원한다

`shared_ptr`과 `unique_ptr`의 관계를 살펴보자

```

1 int main() {
2     shared_ptr<int> sp( new int );
3     unique_ptr<int> up( new int );
4
5     shared_ptr<int> sp1 = up;           // error
6     shared_ptr<int> sp2 = std::move(up); // ok
7
8     shared_ptr<int> up1 = sp;           // error
9     shared_ptr<int> up2 = std::move(sp); // error
10 }

```

```

1 class Shape {};
2 class Rect : public Shape {};
3 class Circle : public Shape {};
4
5 Shape* CreateShape(int type) {
6     Shape* p = nullptr;
7     switch(type) {
8         case 1 : p = new Rect; break;
9         case 2 : p = new Circle; break;
10    }
11    return p;
12 }
13
14 int main() {
15     Shape* p = CreateShape(1);
16 }

```

-
- 위 코드에서 Shape*를 반환하는 대신 스마트 포인터를 사용하고 싶다면 shared_ptr과 unique_ptr 중 어느 것을 사용해야 좋을까?
 - unique_ptr을 반환 타입으로 하는 것이 좋다. unique_ptr은 shared_ptr, unique_ptr 타입으로 모두 받을 수 있기 때문이다

```

1 class Shape {};
2 class Rect : public Shape {};
3 class Circle : public Shape {};
4
5 unique_ptr<Shape> CreateShape(int type) {
6     unique_ptr<Shape> p;
7     switch(type) {
8         case 1 : p.reset(new Rect); break;
9         case 2 : p.reset(new Circle); break;
10    }
11    return p;
12 }
13
14 int main() {
15     unique_ptr<Shape> p1 = CreateShape(1);
16     shared_ptr<Shape> p2 = CreateShape(1);
17 }
```

3.6.7 chrono (ratio, duration)

chrono 라이브러리의 첫번째 예시로 std::ratio에 대해 살펴보자

```

1 template<intmax_t _Nx, intmax_t _Dx = 1>
2 struct ratio {
3     static constexpr intmax_t num = _Nx;
4     static constexpr intmax_t den = _Dx;
5
6     typedef ratio<num, den> type;
7 };
8
9 int main() {
10     std::ratio<2, 4> r1;           // 2/4 ==> 약분되어 1/2 보관
11
12     cout << sizeof(r1) << endl;    // 1 (크기가 없음)
13
14     cout << r1.num << endl;        // 1
15     cout << r1.den << endl;        // 2
16
17     cout << std::ratio<2,4>::num << endl; // 1
18     cout << std::ratio<2,4>::den << endl; // 2
19 }
```

- ratio 템플릿은 컴파일 시간 분수 값을 나타내는 템플릿이다
- <ratio> 헤더 파일이 필요하다

std::ratio 연산에 대해 살펴보자

```

1 int main() {
2     ratio_add< ratio<1,4>, ratio<2,4> > r2; // 1/4 + 2/4 = 3/4
3
4     cout << r2.num << endl; // 3
5     cout << r2.den << endl; // 4
6
7     // 자주 사용하는 연산
8     typedef ratio<1, 1000> milli;
9     typedef ratio<1000, 1> kilo;
10
11     milli m;
12     kilo k;
```

```
13
14 cout << k.num << endl; // 1000
15 cout << k.den << endl; // 1
16 }
```

-
- 대수 연산: ratio_add, ratio_subtract, ratio_multiply, ratio_divide
 - 비교 연산: ratio_equal, ratio_less, ratio_greater, ratio_not_equal, ratio_less_equal, ratio_greater_equal
 - 자주 사용하는 연산: centi, milli, micro, kilo, mega, giga

std::chrono::duration 개념에 대해 살펴보자

```
1 int main() {
2     double distance = 3;                                // 3m, 3km, 3cm인지 알 수 없다
3
4     std::chrono::duration<double, ratio<1,1>> d1(3);    // m
5     std::chrono::duration<double, milli> d2(d1);        // mm
6     std::chrono::duration<double, kilo> d3(d1);         // km
7
8     cout << d2.count() << endl;                          // 3000
9     cout << d3.count() << endl;                          // 0.003
10
11
12 using Meter = std::chrono::duration<double, ratio<1,1>>;
13 using MilliMeter = std::chrono::duration<double, milli>;
14 using KiloMeter = std::chrono::duration<double, kilo>;
15
16
17 Meter m(3);
18 MilliMeter mm(m);          // 3000
19 KiloMeter km(m);           // 0.003
20 }
```

-
- duration으로 값과 단위를 한 변수에 저장할 수 있다

duration과 캐스팅의 관계에 대해 살펴보자

```
1 int main() {
2     using MilliMeter = duration<int, milli>;
3     using KiloMeter = duration<int, kilo>;
4     using Meter = duration<int, ratio<1,1>>;
5
6     Meter m(600);
7     MilliMeter mm(m);      // ok. 600000
8     KiloMeter km(m);       // error! 0.6 ==> 0 또는 1이 되어 데이터 손실 발생
9 }
```

-
- 데이터 손실이 발생하기 때문에 컴파일 에러가 발생한다

```
1 KiloMeter km = duration_cast<KiloMeter>(m);
2
3 KiloMeter km = round<KiloMeter>(m);
4
5 KiloMeter km = floor<KiloMeter>(m);
```

-
- 위와 같이 작성해야 컴파일 에러가 발생하지 않는다. 하지만 duration_cast는 값을 반올림, 버림, 올림할지 선택할 수 없다
 - c++17에 들어오면서 floor, ceil, round, abs가 등장하여 반올림, 버림, 올림, 절대값 등을 표현할 수 있게 되었다

시간 관련 타입들을 살펴보자

```
1 int main() {
2     using seconds = duration<int, ratio<1,1>>;
3     using minutes = duration<int, ratio<60>>;
4     using hours = duration<int, ratio<3600>>;
```

```

5  using milliseconds = duration<int, milli>;
6
7  hours h(1);
8  minutes m(h); // 60
9  seconds s(h); // 360
10
11 hours h2(s); // error! 작은 타입이 큰 타입에 들어가면 캐스팅 에러가 발생한다
12 hourse h2 = duration_cast<hours>(s);
13
14 using days = duration<int, ratio<3600*24, 1>>;
15 days d(1);
16 minutes m2(d); // 60 * 24. 하루는 1440분
17 }
```

- 시간 타입들은 이미 C++ 표준에 구현되어 있다. 하지만 날짜 타입들은 정의되어 있지 않다. 이를 사용하고 싶으면 직접 정의해서 사용해야 한다

duration 객체의 초기화 방법에 대해 살펴보자

```

1 void foo( seconds s ) {}
2
3 int main() {
4     seconds s1(3);           // ok. 직접 초기화
5     seconds s2 = 3;          // error. 복사 초기화는 불가능
6
7     seconds s3 = 3s;         // ok. seconds operator""s(3)이 호출되어 초기화 가능
8     seconds s4 = 3min;       // ok. 180
9     seconds s5 = 3min + 40s; // ok. 연산 가능
10
11    foo( 3 );              // error
12    foo( 3s );              // ok
13
14    this_thread::sleep_for( 3s ); // ok
15 }
```

- duration은 explicit 생성자이므로 직접 초기화(direct initialization)은 가능하지만 복사 초기화(copy initialization)은 불가능하다

- 시간 관련 유저 정의 리터럴(literal)의 종류: operator""h, operator""min, operator""s, operator""ms, operator""us, operator""ns

STL을 사용하여 현재 시간을 구하는 방법에 대해 살펴보자

```

1 int main() {
2     system_clock::time_point tp = system_clock::now(); // 현재 시간을 time_point 타입으로 반환
3
4     // 1. 현재 시간 단위 변환
5     nanoseconds ns = tp.time_since_epoch(); // ns
6     hours h = duration_cast<hours>(ns);      // h
7
8     // 2. 현재 날짜 출력 : time_point ==> string 타입 변환
9     time_t t = system_clock::to_time_t(tp);
10    string s = ctime(&t);
11    cout << s << endl;                      // 현재 날짜 출력
12 }
```

- epoch_time : 1970년 1월 1일 0시를 기점으로 경과된 시간 단위

4 Design Pattern

4.1 Separation of commonality and variability

4.1.1 Template method pattern

행위 패턴(behavior pattern) 중 하나인 template method 패턴은 오퍼레이션에는 알고리즘의 처리 과정만을 정의하고 각 단계에서 수행할 구체적인 처리는 서브클래스에서 정의하는 패턴을 말한다. 다시 말하면 template method 패턴은 "알고리즘의 처리 과정은 변경하지 않고 알고리즘 각 단계의 처리를 서브클래스에서 재정의" 할 수 있게 한다.

```
1 class AbstractClass {
2     void templateMethod();
3     void primitive1();
4     void primitive2();
5 };
6
7 class ConreateClass : public AbstractClass {
8     void primitive1();
9     void primitive2();
10};
```

- template method 패턴은 일반적으로 위와 같은 형태를 띤다

```
1 class PainterPath {
2     public:
3     void begin(){}
4     void end(){}
5
6     void draw_rect() { std::cout << "draw rect" << std::endl; }
7     void draw_circle() { std::cout << "draw circle" << std::endl; }
8 };
9
10 class Painter {
11     public:
12     void draw_path(const PainterPath& path) {}
13 };
14
15 //-----
```

```
17 class Shape {
18     public:
19     virtual ~Shape() {}
20     virtual void draw() = 0;
21 };
22
23 class Rect : public Shape {
24     public:
25     void draw() override {
26         PainterPath path;
27         path.begin();
28
29         // path 멤버 함수로 그림을 그린다
30         path.draw_rect();
31
32         path.end();
33
34         Painter surface;
35         surface.draw_path(path);
36     }
37 };
38
39 class Circle : public Shape {
40     public:
41     void draw() override {
```

```

42     PainterPath path;
43     path.begin();
44
45     // path 멤버 함수로 그림을 그린다
46     path.draw_circle();
47
48     path.end();
49
50     Painter surface;
51     surface.draw_path(path);
52 }
53 };
54
55 int main() {
56     Shape* s1 = new Rect;
57     Shape* s2 = new Circle;
58
59     s1->draw();
60     s2->draw();
61 }
```

- GUI 환경에서 윈도우에 그림을 그리기 위해 대부분의 라이브러리에는 그림을 그리는 클래스를 제공한다. 또한 화면 깜빡임 방지(flicker free)를 하기 위해 더블 버퍼링과 같은 다양한 방법을 제공한다.
- Rect, Circle의 draw 함수를 보면 PainterPath를 호출하는 함수의 대부분의 유사하다

증복된 코드들을 제거해보자.

```

1 class Shape {
2     public:
3     virtual ~Shape() {}
4
5     void draw() {
6         PainterPath path;
7         path.begin();
8
9         draw_imp(path);
10
11     path.end();
12
13     Painter surface;
14     surface.draw_path(path);
15 }
16
17 protected:
18     virtual void draw_imp(PainterPath& path) = 0; // 그림을 그리는 새로운 가상함수를 만든다
19 };
20
21 class Rect : public Shape {
22     public:
23     void draw_imp(PainterPath& path) override {
24         path.draw_rect();
25     }
26 };
27
28 class Circle : public Shape {
29     public:
30     void draw_imp(PainterPath& path) override {
31         path.draw_circle();
32     }
33 };
34
35 int main() {
36     Shape* s1 = new Rect;
37     Shape* s2 = new Circle;
38
39     s1->draw();
```

```
40     s2->draw();
41 }
```

-
- 변하지 않는 코드와 변하는 코드를 찾고 **변해야 하는 코드는 가상함수로 분리**한다
 - 파생 클래스는 알고리즘의 처리 과정을 물려 받으면서 가상함수 재정의를 통해서 **변경이 필요한 부분만 다시 만들 수 있다**

위 클래스에서 `draw` 함수에 template method 패턴이 적용되었다. Template method 패턴은 다른 패턴을 사용할 때도 자주 같이 사용되는 기본적인 패턴 중 하나이므로 기억해야 할 패턴이다.

4.1.2 Strategy pattern

행위 패턴(behavior pattern) 중 하나인 전략(strategy) 패턴은 다양한 알고리즘들을 각각 하나의 "클래스로 캡슐화하여 알고리즘의 대체가 가능"하도록 하는 패턴을 말한다. Strategy 패턴을 이용하면 클라이언트와 독립적인 다양한 알고리즘으로 변형할 수 있다. **알고리즘을 바꾸더라도 클라이언트는 아무런 변형을 할 필요가 없다.**

```
1 class Edit{
2     std::string data;
3     public:
4     std::string get_text() {
5         std::cin >> data;
6         return data;
7     }
8 };
9
10 int main() {
11     Edit edit;
12     while(1) {
13         std::string s = edit.get_text();
14         std::cout << s << std::endl;
15     }
16 }
```

- `Edit` 클래스를 사용자에게 입력을 받을 때 사용하는 GUI Widget이라고 생각하자

`Edit`를 이용해서 나이만 입력받고 싶다고 가정하면 숫자만 입력받도록 제한해야 한다.

```
1 std::string get_text() { // 숫자만 입력받도록 제한
2     data.clear();
3
4     while(1){
5         char c = _getch();
6         if(c==13) break; // enter 키
7
8         if(isdigit(c)) {
9             data.push_back(c);
10            std::cout << c;
11        }
12    }
13    std::cout << "\n";
14    return data;
15 }
```

다음으로 `Edit`를 사용하여 주소를 입력받고 싶다고 하자.

- 현재 `Edit`은 숫자만 입력할 수 있다
- 어떻게 디자인하는 것이 가장 좋은 디자인일까?
- `Edit`의 **제한(validation) 정책은 변경될 수 있어야** 한다.
- Validation 정책 변경 방법 1 : 변하는 것을 가상함수로 분리하는 방법 (**template method 패턴**)

```
1 class Edit{
2     std::string data;
```

```

3   public:
4     std::string get_text() {
5       data.clear();
6
7       while(1){
8         char c = _getch();
9         if(c==13 && iscomplete(data)) break;
10
11        if(validate(data, c)) {
12          data.push_back(c);
13          std::cout << c;
14        }
15      }
16      std::cout << "\n";
17      return data;
18    }
19
20 // 입력 형식을 제한하는 함수
21 virtual bool validate(const std::string& data, char c) {
22   return true;
23 }
24
25 // 입력값이 완성되었는지 확인
26 virtual bool iscomplete(const std::string& data) {
27   return true;
28 }
29 };
30
31 // 숫자만 입력받는 클래스
32 class NumEdit : public Edit {
33   int count;
34   public:
35   NumEdit(int count=9999) : count(count) {}
36
37   bool validate (const std::string& data, char c) override {
38     return data.size() < count && isdigit(c);
39   }
40
41   bool iscomplete(const std::string& data) override {
42     return count != 9999 && data.size() == count;
43   }
44 };
45
46 int main() {
47   NumEdit edit(5); // 5자리 숫자만 입력 가능
48
49   while(1) {
50     std::string s = edit.get_text();
51     std::cout << s << std::endl;
52   }
53 }
```

- 언뜻 보기에도 잘 작성된 코드처럼 보인다. 하지만 template method가 최선의 방법일까? 다른 방법은 없을까?

Validation 정책 변경 방법 2 : 변하는 것을 다른 클래스로 분리하는 방법. 인터페이스를 먼저 만들고 Edit에서 약한 결합으로 다양한 Validation 정책 클래스를 사용하는 방법.

```

1 struct IValidator {
2   virtual bool validate(const std::string& data, char c) = 0;
3   virtual bool iscomplete(const std::string& data) { return true; }
4   virtual ~IValidator() {}
5 };
6
7 class DigitValidator : public IValidator {
8   int count;
9   public:
```

```

10     DigitValidator(int count = 9999) : count(count) {}
11
12     bool validate(const std::string& data, char c) override {
13         return data.size() < count && isdigit(c);
14     }
15
16     bool iscomplete(const std::string& data) override {
17         return count != 9999 && data.size() == count;
18     }
19 };
20
21 class Edit {
22     std::string data;
23     IValidator* val = nullptr;
24
25 public:
26     void set_validator(IValidator* p) { val = p; }
27
28     std::string get_text() {
29         data.clear();
30
31         while(1) {
32             char c = _getch();
33
34             if(c == 13 && (val == nullptr || val->iscomplete(data))) break;
35
36             if(val == nullptr || val->validate(data, c)) {
37                 data.push_back(c);
38                 std::cout << c;
39             }
40         }
41         std::cout << "\n";
42         return data;
43     }
44 };
45
46 int main() {
47     Edit edit;
48     DigitValidator v(5);
49     edit.set_validator(&v);
50
51     while(1) {
52         std::string s = edit.get_text();
53         std::cout << s << std::endl;
54     }
55 }
```

지금까지 설명한 Template method 패턴과 strategy 패턴을 정리해보자

- #1 template method: validation 정책을 가상함수로 분리. NumEdit이 validation 정책을 소유하므로 다른 클래스에서 validation 정책을 사용할 수 없다. 또한 실행 시간에 validation 정책을 교체할 수도 없다.
- #2 strategy: validation 정책을 클래스로 분리. 다른 클래스에서 validation 정책을 사용할 수 있다. 실행 시간에 validation 정책을 교체할 수 있다.
- Edit 예제의 경우는 strategy가 더 적합하지만 template method 자체가 나쁜 것은 아니다.

4.1.3 Policy base design

단위 전략 디자인(policy base design) 패턴에 대해 살펴보자. 단위 전략 디자인은 전통적인 객체지향 디자인 패턴 23개 분류에는 포함되지 않지만 C++ 진영에서는 널리 사용되는 기술이다. 특히 STL의 구현에서 많이 사용되었다.

- 단위 전략 디자인은 전략(strategy) 패턴에 성능 향상을 시킨 버전이다

```

1 template<typename T>
2 class vector {
```

```

3     T* ptr;
4     public:
5     void resize(std::size_t newsize) {
6         ptr = new T[newsize];
7         delete[] ptr;
8     }
9 }
10
11 int main() {
12     vector<int> v;
13 }
```

- 위 코드에서 메모리 할당/해지하는 방법을 변경하고 싶은 경우 어떻게 해야 할까?

방법1 : 변하는 것을 가상함수로 변경한다 (template method)

```

1 template<typename T>
2 class vector {
3     T* ptr;
4     public:
5     void resize(std::size_t newsize) {
6         ptr = allocate(size);
7         deallocate(ptr, size);
8     }
9
10    // 기본은 new, delete 버전
11    virtual T* allocate(std::size_t size) { return new T[size]; }
12    virtual void deallocate(T* ptr, std::size_t size) { delete[] ptr; }
13 };
14
15 // malloc 버전
16 template<typename T>
17 class malloc_vector : public vector<T> {
18     public:
19     T* allocate(std::size_t size) override { return static_cast<T*>(malloc(sizeof(T)*size)); }
20     virtual void deallocate(T* ptr, std::size_t size) override { free(ptr); }
21 };
22
23 int main() {
24     vector<int> v;
25     malloc_vector<int> v;
26 }
```

- 단점: 메모리 할당하는 코드를 재사용하기 어렵다. vector 뿐만 아니라 다른 컨테이너들도 메모리 할당 방식을 변경하려면 파생클래스를 만들어야 한다

방법2 : 변하는 것을 다른 클래스로 분리한다 (strategy 패턴)

```

1 class<typename T> struct IAllocator {
2     virtual T* allocate(std::size_t newsize) = 0;
3     virtual void deallocate(T* ptr, std::size_t size) = 0;
4     ~IAllocator() {}
5 };
6
7 template<typename T>
8 class vector {
9     T* ptr;
10    IAllocator<T>* ax = nullptr;
11    public:
12    void set_allocator(IAllocator<T>* p) { ax = p; }
13
14    void resize(std::size_t newsize) {
15        ptr = ax->allocate(size);
16        ax->deallocate(ptr, size);
17    }
18 };
```

```

19
20 // malloc 버전
21 template<typename T>
22 class malloc_allocator<T> : public IAllocator<T> {
23     public:
24     T* allocate(std::size_t size) override { return static_cast<T*>(malloc(sizeof(T)*size)); }
25     void deallocate(T* ptr, std::size_t size) override { free(ptr); }
26 };
27
28 int main() {
29     vector<int> v;
30     malloc_allocator<int> ma;
31     v.set_allocate(&ma);
32 }
```

- 장점: 메모리를 할당하는 코드(정책)을 다른 컨테이너에서도 사용 가능하다
- 단점: 할당/해지를 위한 함수가 가상함수이므로 느리다

방법3 : 메모리 할당/해지를 담은 정책 클래스(메모리 할당기)를 "인터페이스 기반으로 교체하지 말고 템플릿 인자를 사용해서 교체"하는 방법 (policy base design)

```

1 template<typename T, typename Ax = std::allocator<T>>
2 class vector {
3     T* ptr;
4     Ax ax;
5
6     public:
7
8     void resize(std::size_t newsize) {
9         ptr = ax->allocate(size);
10        ax->deallocate(ptr, size);
11    }
12 };
13
14 template<typename T>
15 class malloc_allocator {
16     public:
17     // inline 치환을 사용하여 성능 저하를 없앤다
18     inline T* allocate(std::size_t size) { return static_cast<T*>(malloc(sizeof(T)*size)); }
19     inline void deallocate(T* ptr, std::size_t size) { free(ptr); }
20 };
21
22 int main() {
23     vector<int, malloc_allocator<int>> v;
24 }
```

strategy pattern	인터페이스를 사용해서 교체 가상함수 기반으로 느리다 실행 시간에도 교체 가능하다
policy base design pattern	템플릿 인자를 사용해서 교체 가상함수가 아닌 인라인 치환도 가능하다 실행시간에도 교체 할 수는 없다

실제 STL vector의 allocator를 교체해보자

```

1 // cppreference.com -> Named Requirement -> Allocator 예제 클래스
2 template<class T>
3 struct Mallocator
4 {
5     typedef T value_type;
6
7     Mallocator() = default;
8
9     template<class U>
```

```

10    constexpr Mallocator(const Mallocator <U>&) noexcept {}
11
12    [[nodiscard]] T* allocate(std::size_t n)
13    {
14        if (n > std::numeric_limits<std::size_t>::max() / sizeof(T))
15            throw std::bad_array_new_length();
16
17        if (auto p = static_cast<T*>(std::malloc(n * sizeof(T))))
18        {
19            report(p, n);
20            return p;
21        }
22
23        throw std::bad_alloc();
24    }
25
26    void deallocate(T* p, std::size_t n) noexcept
27    {
28        report(p, n, 0);
29        std::free(p);
30    }
31
32 private:
33    void report(T* p, std::size_t n, bool alloc = true) const
34    {
35        std::cout << (alloc ? "Alloc: " : "Dealloc: ") << sizeof(T) * n
36        << " bytes at " << std::hex << std::showbase
37        << reinterpret_cast<void*>(p) << std::dec << '\n';
38    }
39
40 template<class T, class U>
41 bool operator==(const Mallocator <T>&, const Mallocator <U>&) { return true; }
42
43 template<class T, class U>
44 bool operator!=(const Mallocator <T>&, const Mallocator <U>&) { return false; }
45
46 int main()
47 {
48     std::vector<int, Mallocator<int> v;
49
50     std::cout << "-----" << std::endl;
51
52     v.resize(5);
53
54     std::cout << "-----" << std::endl;
}

```

- 컨테이너의 메모리 할당 방식을 교체하려면 사용자 정의 메모리 할당기를 만들어서 템플릿 인자로 전달하면 된다
- 메모리 할당기를 만들 때 지켜야하는 규칙을 알아야 한다
- 인터페이스로 강제되어 있지 않으므로 cppreference.com 사이트 같은 곳에 문서화 자료를 참고해야 한다(Named requirement)

4.2 Recursive pattern

4.2.1 Composite pattern

컴포지트 패턴 또는 구조 패턴이란 부분과 전체의 계층을 표현하기 위해 **복잡 객체를 트리 구조**로 만드는 패턴을 말한다. 컴포지트 패턴은 클라이언으로 하려면 **개별 객체와 복합 객체를 모두 동일하게 다룰 수 있도록** 한다.

```

1 int main()
2 {
3     printf("1. - 128) * 64 + (' - 128) - 128) * 64 + (' - 128) 꿀\n");
4     printf("2. - 128) * 64 + (' - 128) 꿀 - 128) * 64 + (' - 128) 꿀\n");
5     printf("3. - 128) * 64 + (' - 128) 뼘 - 128) * 64 + (' - 128) 꿀\n");
6
7     int cmd = 0;
8     scanf_s("%d", &cmd);

```

```

8
9     switch(cmd) {
10        case 1: break;
11        case 2: break;
12        case 3: break;
13    }
14 }
```

- C 프로그래밍은 프로그램 구조가 이해하기 쉬우며 메모리 사용량도 작고 빠르지만 확장성이 부족하고 변화에 유연하지 않다
- 새로운 메뉴가 추가되거나 하위 메뉴를 추가해야 한다면 어떻게 해야 할까?

```

1 class MenuItem {
2     std::string title;
3     int id;
4     public:
5     MenuItem(const std::string& title, int id) : title(title), id(id) {}
6
7     std::string get_title() const { return title; }
8
9     void set_title(const std::string& s) { title = s; }
10
11    void command() {
12        std::cout << get_title() << " 메뉴 선택\n";
13        _getch();
14    }
15 };
16
17 int main() {
18     MenuItem m1("김밥", 11);
19     MenuItem m2("라면", 12);
20
21     m1.command();
22 }
```

- 객체지향 프로그래밍에서 프로그램은 **객체들의 집합**이다
- 프로그램의 기본 요소는 **함수**가 아닌 **클래스**이다

PopupMenu 클래스도 만들어보자

```

1 class MenuItem {
2     std::string title;
3     int id;
4     public:
5     MenuItem(const std::string& title, int id) : title(title), id(id) {}
6
7     std::string get_title() const { return title; }
8
9     void set_title(const std::string& s) { title = s; }
10
11    void command() {
12        std::cout << get_title() << " 메뉴 선택\n";
13        _getch();
14    }
15 };
16
17 class PopupMenu{
18     std::string title;
19     std::vector<MenuItem*> v;
20     public:
21     PopupMenu(const std::string& title) : title(title) {}
22
23     void add_menu(MenuItem* m) { v.push_back(m); }
```

```

25     void command() {
26         while(1) {                                // 무한 루프를 돌면서 메뉴 입력 받음
27             system("cls");
28
29             std::size_t sz = v.size();
30             std::size_t i = 0;
31
32             for(MenuItem* p : v) {
33                 std::cout << ++i << ". " << p->get_title() << "\n";
34             }
35
36             std::cout << i + 1 << ". 종료\n";
37             std::cout << "메뉴를 선택해주세요 >> ";
38
39             int cmd;
40             std::cin >> cmd;
41
42             if( cmd == sz + 1 ) break;
43             if( cmd < 1 || cmd > sz + 1 ) continue;
44
45             v[cmd-1]->command();
46         }
47     };
48 };
49
50 int main() {
51     MenuItem m1("김밥", 11);
52     MenuItem m2("라면", 12);
53
54     PopupMenu pm("오늘의 메뉴");
55     pm.add_menu(&m1);
56     pm.add_menu(&m2);
57
58     pm.command(); // 팝업 메뉴 선택
59 }
```

- PopupMenu를 선택했을 때 해야할 일
- 하위 메뉴를 보여주고
- 사용자 선택을 입력 받음

지금까지 작성한 코드에 컴포지트 디자인 패턴을 적용해보자

```

1  class BaseMenu {
2     std::string title;
3     public:
4     BaseMenu(const std::string& title) : title(title) {}
5
6     virtual ~BaseMenu() {}
7
8     std::string get_title() const { return title; }
9
10    void set_title(const std::string& s) { title = s; }
11
12    virtual void command() = 0;
13 };
14
15 class MenuItem : public BaseMenu {
16     int id;
17     public:
18     MenuItem(const std::string& title, int id) : BaseMenu(title), id(id) {}
19
20     void command() {
21         std::cout << get_title() << " 메뉴 선택\n";
22         _getch();
23     }
24 };
```

```

25
26 class PopupMenu : public BaseMenu {
27     std::string title;
28     std::vector<BaseMenu*> v;
29 public:
30     PopupMenu(const std::string& title) : BaseMenu(title) {}
31
32     void add_menu(BaseMenu* m) { v.push_back(m); }
33
34     void command() {
35         while(1) {
36             system("cls");
37
38             std::size_t sz = v.size();
39             std::size_t i = 0;
40
41             for(BaseMenu* p : v) {
42                 std::cout << ++i << ". " << p->get_title() << "\n";
43             }
44
45             std::cout << i + 1 << ". 종료\n";
46             std::cout << "메뉴를 선택해주세요 >> ";
47
48             int cmd;
49             std::cin >> cmd;
50
51             if( cmd == sz + 1 ) break;
52             if( cmd < 1 || cmd > sz + 1 ) continue;
53
54             v[cmd-1]->command();
55         }
56     }
57 };
58
59 int main() {
60     MenuItem m1("참치 김밥", 11);
61     MenuItem m2("소고기 김밥", 12);
62     MenuItem m3("돈까스 김밥", 13);
63
64     MenuItem m4("라면", 21);
65
66     PopupMenu kimbab("김밥류");
67     kimbab.add_menu(&m1);
68     kimbab.add_menu(&m2);
69     kimbab.add_menu(&m3);
70
71     PopupMenu pm("오늘의 메뉴");
72     pm.add_menu(&kimbab); // PopupMenu의 하위 메뉴로 PopupMenu를 넣을 수 있는가?
73     pm.add_menu(&m4);
74
75     pm.command();
76 }

```

- MenuItem 객체와 PopupMenu 객체를 모두 보관하기 위해 BaseMenu 상위 객체가 필요하다
- BaseMenu에서는 title 등의 공통의 특징을 포함하기 위해 순수가상함수를 사용한다

컴포지트 패턴을 제대로 활용한 예시를 보자

```

1 int main() {
2     PopupMenu* root = new PopupMenu("ROOT");
3     PopupMenu* pm1 = new PopupMenu("해상도 변경");
4     PopupMenu* pm2 = new PopupMenu("색상 변경");
5
6     root->add_menu(pm1);
7     root->add_menu(pm2);
8     //pm1->add_menu(pm2); // 구조를 변경하고 싶은 경우

```

```

9
10    pm1->add_menu(new MenuItem("HD", 11));
11    pm1->add_menu(new MenuItem("FHD", 12));
12    pm1->add_menu(new MenuItem("UHD", 13));
13
14    pm2->add_menu(new MenuItem("RED", 21));
15    pm2->add_menu(new MenuItem("GREEN", 22));
16    pm2->add_menu(new MenuItem("BLUE", 23));
17    pm2->add_menu(new MenuItem("BLACK", 24)); // 색을 추가하고 싶은 경우
18
19    root->command();
20}

```

- 메뉴 구조가 마치 트리 구조처럼 계층 관계가 만들어진다
- 컴포지트 패턴의 의도는 부분과 전체의 계층을 표현하기 위해 **복합객체를 트리 구조로 만드는 것이다**
- 새로운 메뉴를 추가하거나 메뉴의 구조를 변경하는 것이 쉽다

```

1 class PopupMenu : public BaseMenu {
2     ...
3     BaseMenu* submenu(int idx) {
4         return v[idx];
5     }
6 };
7
8 int main() {
9     PopupMenu* root = new PopupMenu("ROOT");
10    root->add_menu( new PopupMenu("색상 변경") );
11    root->add_menu( new MenuItem("저장"), 11 );
12
13    // root->submenu(0)->add_menu( new MenuItem("HD", 21) ); // error!
14    static_cast<PopupMenu*>(root->submenu(0))->add_menu( new MenuItem("HD", 21) ); root->command();

```

- **static_cast** 관련 코드가 깊고 복잡하다
- 타입 캐스팅을 안하려면 **BaseMenu**에 **add_menu**, **submenu** 함수가 포함되어 있어야 한다

타입 안정성을 추구하는 클래스 구조는 다음과 같다
이와 달리 일관성을 유지하기 위한 클래스 구조는 다음과 같다.

```

1 class no_implementation {};
2
3 class BaseMenu {
4     ...
5
6     virtual BaseMenu* submenu(int idx) { throw no_implementation(); }
7     virtual void add_menu(BaseMenu* m) { throw no_implementation(); }
8 };
9
10 ...
11
12 class PopupMenu : public BaseMenu {
13     ...
14     BaseMenu* submenu(int idx) {
15         return v[idx];
16     }
17 };
18
19 int main() {
20     PopupMenu* root = new PopupMenu("ROOT");
21     root->add_menu( new PopupMenu("색상 변경") );
22     root->add_menu( new MenuItem("저장"), 11 );
23
24     root->submenu(0)->add_menu( new MenuItem("HD", 21) ); // 에러 발생하지 않음
25     // static_cast<PopupMenu*>(root->submenu(0))->add_menu( new MenuItem("HD", 21) );
26
27

```

```
28     root->command();
29 }
```

-
- 일관성을 중시한 코드 설계 (전통적인 디자인 패턴)

4.2.2 Decorator

데코레이터 패턴은 객체에 **동적으로 서비스를 추가할 수 있게 하는 패턴**이다. 상속을 사용하여 서비스를 추가하는 것보다 유연한 방법으로 새로운 서비스를 추가할 수 있다.

```
1 class Image {
2     std::string image_path;
3     public:
4     Image(const std::string& path) : image_path(path) { ... }
5
6     void draw() const {
7         std::cout << "draw " << image_path << std::endl;
8     }
9 };
10
11 int main() {
12     Image img("www.image.com/car.jpg");
13     img.draw();
14 }
```

- Image 클래스가 그림을 로드해서 화면에 출력하는 클래스라고 하자. 파일 뿐 아니라 인터넷 다운로드도 지원한다
- 로드된 그림에 **액자나 말풍선 등의 효과를 추가**하고 싶다

상속을 사용하면 어떨까? Image 클래스로부터 모든 기능을 물려받고 액자(또는 말풍선)을 그리는 기능을 추가한다

```
1 class Image {
2     std::string image_path;
3     public:
4     Image(const std::string& path) : image_path(path) { ... }
5
6     void draw() const {
7         std::cout << "draw " << image_path << std::endl;
8     }
9 };
10
11 class Frame : public Image {
12     public:
13     using Image::Image;
14
15     void draw() const {
16         // 기존 코드에 액자(=) 코드 추가
17         std::cout << "===== " << std::endl;
18         Image::draw();
19         std::cout << "===== " << std::endl;
20     }
21 };
22
23 class Balloon : public Image {
24     public:
25     using Image::Image;
26
27     void draw() const {
28         // 기존 코드에 액자(=) 코드 추가
29         std::cout << "=====Balloon===== " << std::endl;
30         Image::draw();
31         std::cout << "=====Balloon===== " << std::endl;
32     }
}
```

```

33     };
34
35     int main() {
36         Image img("www.image.com/car.jpg");
37         img.draw();
38
39         Frame frame("www.image.com/car.jpg");
40         frame.draw();
41
42         Balloon balloon("www.image.com/car.jpg");
43         balloon.draw();
44     }

```

- 상속을 사용한 서비스 추가 특징

- 1) **객체가 아닌 클래스에 기능을 추가한 것**

- Image img 코드가 불리는 순간 이미 그림은 로드되었고 img 객체가 관리한다. img라는 객체에 새로운 기능을 추가할 수 없을까? 상속은 Image라는 클래스에 서비스를 추가한다

- 2) **여러 개의 서비스를 중복해서 추가하기 어렵다**

상속이 아닌 포함을 사용한 기능 추가 방법을 살펴보자

```

1  class Image {
2     std::string image_path;
3     public:
4     Image(const std::string& path) : image_path(path) { ... }
5
6     void draw() const {
7         std::cout << "draw " << image_path << std::endl;
8     }
9 };
10
11 class Frame {
12     Image* img;
13     public:
14     Frame(Image* img) : img(img) {}
15
16     void draw() const {
17         // 추가할 기능 작성
18         img->draw();
19     }
20 };
21
22 class Balloon {
23     Image* img;
24     public:
25     Balloon(Image* img) : img(img) {}
26
27     void draw() const {
28         // 추가할 기능 작성
29         img->draw();
30     }
31 };
32
33 int main() {
34     Image img("www.image.com/car.jpg");
35     img.draw();
36
37     Frame frame(&img);
38     frame.draw();
39
40     Balloon balloon(&img);
41     balloon.draw();
42
43     Balloon balloon(&frame); // error!
44 }

```

-
- 데코레이터 패턴의 핵심은 기능을 중첩할 수 있다는 것이다
 - 현재 코드에는 Balloon balloon(&frame) 같은 코드를 작성하지 못한다

객체의 기능을 중복으로 추가하는 방법에 대해 살펴보자

```
1 struct IDraw {
2     virtual void draw() const = 0;
3     virtual ~IDraw() {}
4 };
5
6 class Image : public IDraw {
7     std::string image_path;
8     public:
9     Image(const std::string& path) : image_path(path) { ... }
10
11    void draw() const {
12        std::cout << "draw " << image_path << std::endl;
13    }
14 };
15
16 class Frame : public IDraw {
17     IDraw* img;
18     public:
19     Frame(IDraw* img) : img(img) {}
20
21    void draw() const {
22        // 추가할 기능 작성
23        img->draw();
24    }
25 };
26
27 class Balloon : public IDraw {
28     IDraw* img;
29     public:
30     Balloon(IDraw* img) : img(img) {}
31
32    void draw() const {
33        // 추가할 기능 작성
34        img->draw();
35    }
36 };
37
38 int main() {
39     Image img("www.image.com/car.jpg");
40     img.draw();
41
42     Frame frame(&img);
43     frame.draw();
44
45     Balloon balloon(&img);    // ok!
46     balloon.draw();
47
48     Balloon balloon2(&frame); // ok!
49     balloon2.draw();
50 }
```

-
- 데코레이터 패턴의 핵심은 **기능을 중첩할 수 있다는 것이다.** 기능 추가를 위해 공통의 인터페이스(IDraw)를 생성하였다

데코레이터 패턴은 기능을 중첩하여 추가하기 때문에 인터페이스 상속 방법을 사용하게 되고 기능 클래스에 상속하므로 기능들이 많아질수록 중복되는 코드가 많아진다. 중복된 코드를 방지하기 위해 기능과 관련된 클래스에는 **Decorator** 클래스에 공통된 코드를 추가한다.

```
1 struct IDraw {
2     virtual void draw() const = 0;
```

```

3     virtual ~IDraw() {}
4 };
5
6 class Image : public IDraw {
7     std::string image_path;
8     public:
9     Image(const std::string& path) : image_path(path) { ... }
10
11    void draw() const {
12        std::cout << "draw " << image_path << std::endl;
13    }
14};
15
16 class Decorator : public IDraw {
17     IDraw* img;
18     public:
19     Decorator(IDraw* img) : img(img) {}
20
21    void draw() const { img->draw(); }
22};
23
24 class Frame : public Decorator {
25     public:
26     Frame(IDraw* img) : Decorator(img) {}
27
28    void draw() const {
29        // 추가할 기능 작성
30        Decorator::draw();
31    }
32};
33
34 class Balloon : public Decorator {
35     public:
36     Balloon(IDraw* img) : Decorator(img) {}
37
38    void draw() const {
39        // 추가할 기능 작성
40        Decorator::draw();
41    }
42};
43
44 int main() {
45     Image img("www.image.com/car.jpg");
46     img.draw();
47
48     Frame frame(&img);
49     frame.draw();
50
51     Balloon balloon(&img);
52     balloon.draw();
53
54     Balloon balloon2(&frame);
55     balloon2.draw();
56 }
```

데코레이터 패턴은 구조 패턴(structural pattern)의 한 종류로써 **객체에 동적으로 서비스를 추가할 수 있게 하며 여러 개의 기능을 중복해서 추가할 수 있도록** 한다. 상속을 사용해서 서비스를 추가(클래스에 기능 추가)하는 것보다 유연한 방법으로 새로운 서비스를 추가할 수 있다.

다른 데코레이터 패턴 예제를 살펴보자

```

1 class Stream {
2     virtual void write(const std::string&) = 0;
3     virtual ~Stream() {}
4 }
5
6 class FileStream : public Stream {
```

```

7   FILE* file;
8   public:
9     FileStream(const char* s, const char* mode = "w+") {
10    file = fopen(s, mode);
11 }
12 ~FileStream() { fclose(file); }
13
14 void write(const std::string& s) {
15   std::cout << "write : " << s << std::endl;
16 }
17 };
18
19 // NetworkStream, PipeStream 클래스 구현 코드 생략
20
21 int main() {
22   FileStream fs("a.txt");
23   fs.write("hello");
24
25   NetworkStream ns("127.0.0.1", 4000);
26   ns.write("hello");
27
28   PipeStream ps("named pipe");
29   ps.write("hello");
30 }
```

- FileStream 클래스 : 특정 파일에 입출력을 위한 다양한 기능을 지원하는 클래스
 - NetworkStream, PipeStream 클래스 : 다양한 장치에 입출력을 위한 클래스들
 - 모든 Stream 클래스는 사용법이 동일하게 설계되는 것이 좋다. 공통의 기반 클래스를 설계해야 한다
 - Stream 클래스 : 모든 입출력 Stream 타입의 기반 클래스
- 파일에 데이터를 write할 때 암호화하는 기능이 필요하다고 가정해보자. 어떻게 구현하는 것이 가장 좋을까?

- 방법1: 상속을 사용한 기능의 추가
- 단점: 모든 Stream 클래스의 파생 클래스가 필요하다
- 단점: 기능을 중첩해서 추가하기 어렵다
- 방법2: 포함을 사용한 기능의 추가([데코레이터 패턴](#))
- 장점: 객체에 중첩해서 기능을 추가할 수 있다

```

1 class Stream {
2   virtual void write(const std::string&) = 0;
3   virtual ~Stream() {}
4 }
5
6 class FileStream : public Stream {
7   FILE* file;
8   public:
9     FileStream(const char* s, const char* mode = "w+") {
10    file = fopen(s, mode);
11 }
12 ~FileStream() { fclose(file); }
13
14 void write(const std::string& s) {
15   std::cout << "write : " << s << std::endl;
16 }
17 };
18
19 class ZipDecorator : public Stream {
20   Stream* stream;
21   public:
22     ZipDecorator(Stream* s) : stream(s) {}
23
24     void write(const std::string& data) {
25       auto s = data + " Zipped";
26 }
```

```

26     stream->write(s);
27 }
28 };
29
30 class CryptoDecorator : public Stream {
31     Stream* stream;
32     public:
33     CryptoDecorator(Stream* s) : stream(s) {}
34
35     void write(const std::string& data) {
36         auto s = data + " Crypto";
37         stream->write(s);
38     }
39 };
40
41 int main() {
42     FileStream fs("a.txt");
43
44     CryptoDecorator cd(&fs); // 암호화
45
46     ZipDecorator zd(&cd); // 압축+암호화
47     zd.write("hello");
48 }
```

- C# 언어의 Stream 클래스가 대표적으로 데코레이터 패턴이 구현된 코드이다

4.3 Principle of indirect layer

4.3.1 Adapter

벽에 220V의 전원이 있다고 하자. 노트북은 12V의 전원이 공급되어야 작동이 된다고 할 때 노트북의 어댑터는 220V의 전압을 12V로 변환해주는 역할을 수행한다. 디자인 패턴에서 어댑터 패턴(adapter pattern)은 [클라이언트가 기대하는 형태의 인터페이스로 변환해주는 패턴](#)을 말한다.

```

1  class CoolText {
2      std::string text;
3      std::string font_name;
4      std::size_t font_size;
5      // ...
6      public:
7      CoolText(const std::string& text) :text(text) { }
8
9      void show() { std::cout << text << std::endl; }
10 }
```

- CoolText 클래스 : 문자열을 보관했다가 화면에 출력해주는 클래스. 문자열을 예쁘게 출력할 수 있는 다양한 기능을 제공한다 (예전부터 가지고 있었던 클래스라고 가정한다)

```

1  class Shape {
2      public:
3      virtual ~Shape() {}
4      virtual void draw = 0;
5  };
6
7  class Rect : public Shape {
8      public:
9      void draw() override { std::cout << "draw Rect" << std::endl; }
10 }
11
12 class Circle : public Shape {
13     public:
14     void draw() override { std::cout << "draw Circle" << std::endl; }
15 }
```

- 도형편집기 코드에 Rect, Circle 외에 "문자열을 편집하여 출력하는 클래스를 추가"하고 싶다
- 예전부터 가지고 있던 CoolText를 활용할 수는 없을까?
- 도형편집기에 사용하려면 반드시 Shape 클래스로부터 상속받아야 하고 draw() 함수를 제공해야 한다
- CoolText 클래스의 show() 함수를 draw() 함수로 변환하는 어댑터 클래스가 필요하다

```

1 class ClsAdapter : public CoolText, public Shape {
2     public:
3         ClsAdapter(const std::string& text) : CoolText(text) {}
4
5         void draw() override { CoolText::show(); }
6     };
7
8     int main() {
9         std::vector<Shape*> v;
10
11         v.push_back( new ClsAdapter("Hello") );
12         v[0]->draw();
13     }

```

- CoolText가 가진 문자열 관리, 출력 기능을 모두 물려받고 도형편집기 시스템의 요구사항도 만족한다 ([어댑터 패턴](#))

어댑터 패턴의 종류에 대해 살펴보자

```

// 클래스 어댑터
1 class ClsAdapter : public CoolText, public Shape {
2     public:
3         ClsAdapter(const std::string& text) : CoolText(text) {}
4
5         void draw() override { CoolText::show(); }
6     };
7
8
// 객체 어댑터
9 class ObjAdapter : public Shape {
10     CoolText* ct; // 이미 생성된 CoolText 객체를 가리킨다
11     public:
12         ObjAdapter(CoolText* ct) : ct(ct) {}
13
14         void draw() override { ct->show(); }
15     };
16
17
int main() {
18     std::vector<Shape*> v;
19
20     CoolText ct("Hello"); // CoolText: 클래스
21     // ct: 객체
22
23     v.push_back(&ct); // error!
24     v.push_back( new ObjAdapter(&ct) ); // ok
25
26     v[0]->draw();
27 }

```

- 1) 클래스 어댑터 : 클래스의 인터페이스를 변경(다중 상속 형태)
- 2) 객체 어댑터 : 객체의 인터페이스를 변경(포함 형태)

STL과 어댑터 패턴에 대해 살펴보자

```

1 int main() {
2     stack<int> s;
3     s.push(10);
4     s.push(20);
5 }

```

- c++ 표준 라이브러리인 STL에도 어댑터 패턴을 사용한 설계가 많이 있음(container adapter, iterator adapter,

range adapter, ...)

- STL에는 다양한 시퀀스 컨테이너들이 제공된다 (`std::vector`, `std::list`, `std::deque`)
- STL에는 `std::stack`도 제공된다. 이를 어떻게 구현했을까?
- 방법1 : `stack`의 모든 기능을 직접 구현하는 방법
- 단점 : 메모리 관리 등의 모든 기능을 직접 구현해야 한다.
- 단점 : 코드 메모리도 증가하게 된다
- 방법2 : 기존에 존재하는 시퀀스 컨테이너의 함수 이름만 변경해서 `stack`처럼 보이게 한다 (어댑터 패턴)
- STL에서는 어댑터 패턴을 사용하여 `std::stack`을 구현하였다

어댑터 패턴을 사용하여 `std::stack`을 구현해보자. 우선 상속을 사용한 어댑터 패턴을 구현해보자

```
1 // 상속을 사용한 어댑터 패턴
2 template<typename T>
3 class stack : public std::list<T> {
4     public:
5         void push(const T& a) {
6             std::list<T>::push_back(a);
7         }
8
9         void pop() {
10            std::list<T>::pop_back();
11        }
12
13     T& top() {
14         return std::list<T>::back();
15     }
16 };
17
18 int main() {
19     stack<int> s;
20     s.push(10);
21     s.push(20);
22 }
```

- 단점 : `std::list`로부터 `push_front` 등의 함수도 물려받게 된다(`s.push_front(100)`과 같은 함수도 호출된다)
- 자바 언어의 `stack` 디자인은 `vector`로부터 상속을 받아 구현되어 있어서 안좋은 디자인 패턴으로 유명하다

다음으로 `private` 상속을 사용한 스택 어댑터 패턴에 대해 살펴보자

```
1 // private 상속을 사용한 어댑터 패턴
2 template<typename T>
3 class stack : private std::list<T> {
4     public:
5         void push(const T& a) {
6             std::list<T>::push_back(a);
7         }
8
9         void pop() {
10            std::list<T>::pop_back();
11        }
12
13     T& top() {
14         return std::list<T>::back();
15     }
16 };
17
18 int main() {
19     stack<int> s;
20     s.push(10);
21     s.push(20);
22 }
```

- 기본 클래스의 멤버 함수를 파생 클래스의 내부에서만 사용한다(외부에 노출하지 않음)
- 구현은 물려받지만 인터페이스는 물려받지 않겠다는 의미
- C++ 진영에서 가끔 볼 수 있는 디자인으로 다른 언어에는 `private` 상속이 존재하지 않기 때문에 자주 사용되지는

않는다

포함을 사용한 어댑터 패턴에 대해 살펴보자

```
1 // 포함을 사용한 어댑터 패턴
2 template<typename T>
3 class stack : {
4     std::list<T> c;
5     public:
6     void push(const T& a) {
7         c.push_back(a);
8     }
9
10    void pop() {
11        c.pop_back();
12    }
13
14    T& top() {
15        return c.back();
16    }
17 };
18
19 int main() {
20     stack<int> s;
21     s.push(10);
22     s.push(20);
23 }
```

- **private** 상속 : `std::list`에 가상함수가 있다면 `stack`이 `override`할 수 있다
- 포함 : `std::list`에 가상함수를 `stack`이 `override`할 수 없다

포함을 사용한 `stack`을 조금 더 발전시켜보자

```
1 template<typename T, typename C = std::deque<T>>
2 class stack : {
3     C c;
4     public:
5     constexpr void push(const T& a) { c.push_back(a); }
6     constexpr void pop() { c.pop_back(); }
7     constexpr T& top() { return c.back(); }
8 };
9
10 int main() {
11     stack<int, std::vector<int>> s;
12     stack<int, std::list<int>> s;
13     s.push(10);
14     s.push(20);
15 }
```

- `std::stack`을 만들 때 반드시 `std::stack`을 사용해야 할까? (`std::vector`, `std::deque`는 안될까?)
- 사용자가 선택할 수 있게 하면 어떨까? 단위 전략 디자인(policy base design) 패턴을 사용하면 클래스가 사용하는 정책을 템플릿 인자를 통해서 교체할 수 있게 된다
- 위 코드가 STL의 `std::stack`과 가장 유사하다(`std::stack`을 container adapter라고 부른다)
- `std::stack`은 클래스 어댑터일까 객체 어댑터일까? 클래스 어댑터이다

STL의 반복자 어댑터(iterator adapter)라는 개념에 대해 살펴보자

```
1 int main() {
2     std::vector<int> s = {1,2,3,4,3,2,1};
3
4     auto first = s.begin();
5     auto last = s.end();
6
7     auto ret = std::find(first, last, 3);
8 }
```

- `std::find`로 검색 시 뒤에서부터 검색할 수 없을까?
- `std::find`는 첫번째 인자로 받은 반복자를 ++로 이동하면서 검색한다
- 방법1: 거꾸로 동작하는 반복자를 새롭게 만들자 : `vector`, `list`, `deque` 등 모든 컨테이너에 새로운 반복자를 만들어어야 하므로 쉽지 않다
- 방법2: ++ 연산시 기존 반복자에 -- 연산을 수행하는 어댑터를 만들자([어댑터 패턴](#))

```

1 int main() {
2     std::vector<int> s = {1,2,3,4,3,2,1};
3
4     std::reverse_iterator first(s.end());
5     std::reverse_iterator last(s.begin());
6
7     auto ret = std::find(first, last, 3);
8 }
```

- `std::reverse_iterator` : 기존 반복자의 동작을 거꾸로 수행하는 반복자 어댑터
- 이외에도 STL에는 많은 반복자 어댑터들이 존재한다(cppreference 참조)

4.3.2 Proxy

구조 패턴(structural pattern) 중의 하나인 프록시 패턴은 다른 객체에 접근하기 위해 중간 대리 역할을 하는 객체를 두는 패턴을 말한다.

```

1 class Image {
2     std::string name;
3     public:
4     Image(const std::string& name) : name(name) {
5         std::cout << "open " << name << '\n';
6     }
7     void draw() { std::cout << "draw " << name << '\n'; }
8
9     int width() const { return 100; }
10    int height() const { return 100; }
11 };
12
13 int main() {
14     Image* img = new Image("c:\\a.png");
15     img->draw();
16
17     int w = img->width();
18     int h = img->height();
19 }
```

- `Image` 클래스 : 그림 파일을 로드하여 관리하는 클래스
- `Image`를 그릴 필요는 없고 크기 정보만 없고 싶은 경우 그림 파일 전체를 메모리에 로드할 필요가 없다
- 그림 파일의 헤더 부분에서 읽어 올 수 있다
- 크기 정보만 얻고 싶으면 [지연된 생성을 위한 대행자\(delegator\)](#)를 만들어야 한다([프록시 패턴](#))
- 필요한 경우(draw)에만 `Image` 객체를 생성하면 된다

```

1 class Image {
2     std::string name;
3     public:
4     Image(const std::string& name) : name(name) {
5         std::cout << "open " << name << '\n';
6     }
7     void draw() { std::cout << "draw " << name << '\n'; }
8
9     int width() const { return 100; }
10    int height() const { return 100; }
11 };
12
13 class ImageProxy {
```

```

14     std::string name;
15     Image* img = nullptr;
16     public:
17     ImageProxy(const std::string& name) : name(name) {}
18     int width() const { return 100; } // 헤더에서 정보 획득
19     int height() const { return 100; } // 헤더에서 정보 획득
20
21     void draw() { // 필요한 경우에만 draw 사용
22         if(img == nullptr) {
23             img = new Image(name);
24         }
25         img->draw();
26     }
27 };
28
29 int main() {
30     ImageProxy* img = new ImageProxy("c:\\a.png");
31
32     int w = img->width();
33     int h = img->height();
34
35     img->draw();
36 }
```

- 지연된 생성
- 보안(인증)의 추가
- 기능의 추가(로그 기록)
- 원격지 서버에 대한 대행자

프록시 패턴에 인터페이스를 추가하여 동일한 메소드를 호출하도록 할 수 있다

```

1 struct IImage {
2     ...
3 };
4
5 class Image : public IImage {
6     ...
7 };
8
9 class ImageProxy : public IImage {
10    ...
11 };
12
13 int main() {
14     IImage* img = new ImageProxy("c:\\a.png");
15
16     int w = img->width();
17     int h = img->height();
18
19     img->draw();
20 }
```

4.3.3 Facade

구조 패턴(structural pattern) 중의 하나인 파사드 패턴은 서브 시스템을 합성하는 다수 객체들의 인터페이스 집합에 대해 "일관된 하나의 인터페이스를 제공"할 수 있도록 하는 디자인 패턴이다. 파사드 패턴은 서브 시세들을 "사용하기 쉽게 하기 위한 포괄적 개념의 인터페이스를 정의"한다

```

1 int main() {
2     WSADATA w;
3     WSAStartup(0x202, &w);
4
5     int sock = socket(PF_INET, SOCK_STREAM, 0);
6 }
```

```

7   struct sockaddr_in addr = { 0 };
8     addr.sin_family = AF_INET;
9     addr.sin_port = htons(4000);
10    addr.sin_addr.s_addr = inet_addr("127.0.0.1");
11
12    bind(sock, (SOCKADDR*)&addr, sizeof(addr));
13
14    listen(sock, 5);
15
16    struct sockaddr_in addr2 = { 0 };
17    int sz = sizeof(addr2);
18
19    accept(sock, (SOCKADDR*)&addr2, &sz);
20
21    WSACleanup();
22 }

```

- c언어로 만든 TCP 서버 프로그램
- c언어는 데이터와 함수가 분리되어 있고 생성자, 소멸자의 문법도 없다
- 따라서 코드가 복잡해 보인다
- c++ 같은 객체 지향 언어에서는 네트워크 프로그램에 사용되는 다양한 개념을 각각 클래스로 설계한다

```

1 class NetworkInit {
2   public:
3     NetworkInit() {
4       WSADATA w;
5       WSAStartup(0x202, &w);
6     }
7     ~NetworkInit() { WSACleanup(); }
8   };
9
10 // IP 주소를 관리하는 클래스
11 class IPAddress {
12   SOCKADDR_IN addr;
13   public:
14     IPAddress(const char* ip, short port) {
15       addr.sin_family = AF_INET;
16       addr.sin_port = htons(port);
17       addr.sin_addr.s_addr = inet_addr(ip);
18     }
19
20     SOCKADDR* getRawAddress() {
21       return (SOCKADDR*)&addr;
22     }
23   };
24
25 // Socket 작업을 책임지는 클래스
26 class Socket {
27   int sock;
28   public:
29     Socket(int type) { sock = socket(PF_INET, type, 0); }
30
31     void Bind(IPAddress* ip) {
32       ::bind(sock, ip->getRawAddress(), sizeof(SOCKADDR_IN));
33     }
34
35     void Listen() { ::listen(sock, 5); }
36
37     void Accept() {
38       struct sockaddr_in addr2 = { 0 };
39       int sz = sizeof(addr2);
40       accept(sock, (SOCKADDR*)&addr2, &sz);
41     }
42   };
43

```

```

44 int main() {
45     NetworkInit init;
46     Socket sock(SOCK_STREAM);
47
48     IPAddress addr("127.0.0.1", 4000);
49     sock.Bind(&addr);
50     sock.Listen();
51     sock.Accept();
52 }
```

- 네트워크 프로그래밍 클래스는 어렵지만 라이브러리 작성자들이 작성한 코드이므로 반드시 알 필요는 없다
- 유저 입장에서는 `main()` 내부의 코드만 이해하면 되므로 C언어로 작성한 것보다는 간편해진다
- 하지만 실제로 위 코드를 사용하는게 간편할까? 조금 더 편한 방법은 없을까?

`TCPServer` 클래스를 만들어보자

- TCP Server를 만드는데 필요한 "일련을 절차에 대한 포괄적 개념의 인터페이스를 제공"하는 클래스
- 사용하기 쉽게 하기 위한 클래스

```

1 class TCPServer {
2     NetworkInit init;
3     Socket sock{SOCK_STREAM};
4     public:
5     void Start(const char* ip, short port) {
6         IPAddress addr(ip, port);
7         sock.Bind(&addr);
8         sock.Listen();
9         sock.Accept();
10    }
11 };
12
13 int main() {
14     TCPServer server;
15     server.start("127.0.0.1", 4000);
16 }
```

- 네트워크 프로그래밍을 알 필요없이 코드가 훨씬 간단해진다
- low level의 과정을 포괄적 개념의 인터페이스를 제공함으로써 사용하기 쉽게 해주는 패턴([파사드 패턴](#))

Facade	서브 시스템 사용을 쉽게 하는 포괄적 개념의 인터페이스
2차 API	객체지향, 클래스, 1차 API의 각각의 기능에 1:1로 대응하는 클래스
1차 API	C언어 기반, 각각의 분야(네트워크, 음악, 영상 등)의 합수와 데이터
OS	Windows, Linux, OSX 등... 대부분 C언어로 작성된 OS

4.3.4 Bridge

행위 패턴(Behavior pattern) 중 하나인 브릿지 패턴에 대해 알아보자

- 특정 기능을 구현한 클래스 A를 많은 사용자들이 사용하고 있다고 가정하자. 만약 A의 업데이트가 자주 일어나는 경우 사용자 코드도 모두 변경되어야 한다. [사용자와 상관없이 구현만 독립적으로 업데이트할 수 있을까?](#)
- A 클래스와 사용자 사이에 간접층을 넣어서 [구현과 추상을 독립적으로 업데이트할 수 있게 하는 패턴](#)을 브릿지 패턴이라고 한다.

```

1 class IPod {
2     public:
3     void play() { std::cout << "Play MP3 with IPod" << std::endl; }
4     void stop() { std::cout << "Stop" << std::endl; }
5 };
6
```

```

7   class People {
8     public:
9       void use(IPod* p) {
10      p->play();
11      p->stop();
12    }
13  };
14
15 int main() {
16   People p;
17   IPod pod;
18   p.use(&pod);
19 }
```

- People의 use() 함수에서 IPod을 직접 사용하고 있다
- 두 클래스 사이에 강한 결합이 형성되어 다른 제품으로 교체가 불가능하다
- 인터페이스를 생성하여 결합을 느슨하게 해준다

```

1  class IMP3() {
2    virtual void play() = 0;
3    virtual void stop() = 0;
4    virtual ~IMP3() {}
5  };
6
7  class IPod : public IMP3 {
8    public:
9      void play() { std::cout << "Play MP3 with IPod" << std::endl; }
10     void stop() { std::cout << "Stop" << std::endl; }
11   };
12
13 class People {
14   public:
15     void use(IMP3* p) {
16       p->play();
17       p->stop();
18     }
19   };
20
21 int main() {
22   People p;
23   IPod pod;
24   p.use(&pod);
25 }
```

- 인터페이스를 사용하여 결합이 약해졌다
- 1분 미리듣기 기능을 추가하고자 한다 (`play_one_minute()`)
- 사용자가 새로운 기능을 요구하면 IMP3 인터페이스가 수정되어야 한다. 인터페이스의 변경은 모든 제품의 변경을 의미한다
- 반대로 인터페이스의 코드가 바뀌면 모든 사용자의 코드도 변경되어야 한다
- 사용자와 IMP3 구현의 업데이트를 상호 독립적으로 만들 수 없을까? ([브릿지 패턴 사용 이유](#))

```

1 // 구현층
2 class IMP3() {
3   virtual void play() = 0;
4   virtual void stop() = 0;
5   virtual ~IMP3() {}
6 };
7
8 class IPod : public IMP3 {
9   public:
10    void play() { std::cout << "Play MP3 with IPod" << std::endl; }
11    void stop() { std::cout << "Stop" << std::endl; }
12 }
```

```

13
14 // 추상층
15 class MP3 {
16     IMP3* impl;
17     public:
18     MP3(IMP3* p = nullptr) : impl(p) {
19         if(impl == nullptr) {
20             impl = new IPod;
21         }
22     }
23
24     void play() { impl->play(); }
25     void stop() { impl->stop(); }
26
27     void play_one_minute() {
28         impl->play();
29         // 1분 후
30         impl->stop();
31     }
32 };
33
34 class People {
35     public:
36     void use(IMP3* p) {
37         p->play();
38         p->stop();
39     }
40 };
41
42 int main() {
43     People p;
44     MP3 mp3;
45     p.use(&mp3);
46 }
```

- 1분 미리듣기(play_one_minute())를 원하는 경우 구현층(IMP3, IPod)을 손대지 않더라도 추상층(MP3) 내에서 처리할 수 있다

- **PIMPL**(Pointer to IMPLementation) : c++ 진영에서 브릿지 패턴을 나타내는 또 다른 용어 (코드를 보면 IMP3* impl를 사용하고 있기 때문)

c++ 진영에서 널리 사용되는 PIMPL 패턴(= 브릿지 패턴) 예제를 살펴보자. 다수의 파일에서 Point.cpp, Point.h를 사용한다고 하자. Point.h가 업데이트 되는 경우 다수의 파일도 같이 내용을 변경해야 한다.

```

1 // PointImpl.h
2 class PointImpl {
3     int x,y;
4     public:
5     PointImpl(int x, int y);
6     void print() const;
7 }



---


1 // PointImpl.cpp
2 PointImpl::PointImpl(int x, int y) : x(x), y(y) {}
3 void PointImpl::print() const {
4     std::cout << x << ", " << y << std::endl;
5 }



---


1 // Point.h
2 class PointImpl; // 핵심! 전방 선언으로 충분하다
3
4 class Point {
5     PointImpl* impl;
6     public:
7     Point(int x, int y);
```

```
8     void print() const;
9 }
```

```
1 // Point.cpp
2 #include <Point.h>
3 #include <PointImpl.h>
4
5 Point::Point(int x, int y) : impl(new PointImpl(x,y)) {}
6
7 void Point::print() const {
8     impl->print();
9 }
```

```
1 // main.cpp
2 int main() {
3
4 }
```

- PointImpl(구현층)과 Point(추상층)을 분리한다
- 핵심은 Point.h 파일에 PointImpl.h를 추가하지 않아도 된다는 것이다. (Point.cpp, PointImpl.cpp, main.cpp에서만 사용)
- PointImpl.h의 수정된다고 하더라도 Point.h을 변경할 필요가 없어진다

PIMPL 기술의 장점

- 컴파일 속도가 빨라진다 (컴파일 방화벽)
- 완벽한 정보 은닉을 할 수 있다

4.4 Notification, enumeration, visitation

4.4.1 Observer

엑셀에는 데이터와 차트를 연결할 수 있는 기능이 있다. 차트는 여러 차트를 동시에 표현할 수 있으므로 데이터와 차트의 관계는 1:N의 관계가 된다. 데이터가 업데이트되면 차트는 자동으로 업데이트 된다. 이러한 기능은 옵저버 패턴의 대표적인 예시이다.

행위 패턴(behavior pattern)의 한 종류인 옵저버 패턴은 객체 사이의 1:N 종속성을 정의하고 한 객체의 상태가 변하면 종속된 다른 객체들에 통보가 가고 자동으로 수정이 일어나게 하는 패턴을 말한다.

```
1 class Table {
2     void edit(){
3         while(1){
4             std::cout << "data >> ";
5             std::cin >> data;
6         }
7     }
8 };
9
10 int main() {
11     Table t;
12     t.edit(); // table 편집 모드
13 }
```

- 테이블의 데이터가 변경되면 연결된 모든 차트가 수정되어야 한다
- 방법1: 차트에서 루프를 돌면서 테이블이 변하는지 관찰한다
- 차트가 여러개인 경우 모든 차트에서 루프를 돌면서 테이블을 관찰해야 한다
- 방법2: 차트를 테이블에 등록하고 테이블이 수정되면 등록된 차트에 통보한다 (옵저버 패턴의 핵심)

```
1 struct IChart {
2     virtual void update(int n) = 0;
3     virtual ~IChart() {}
4 };
5
```

```

6   class BarChart : public IChart {
7     public:
8       void update(int n) override {
9         std::cout << "Bar Chart : ";
10        for(int i=0; i<n; i++){
11          std::cout << "*";
12        }
13        std::cout << std::endl;
14      }
15    };
16
17  class PieChart : public IChart {
18    public:
19      void update(int n) override {
20        std::cout << "Pie Chart : ";
21        for(int i=0; i<n; i++){
22          std::cout << "(";
23        }
24        std::cout << std::endl;
25      }
26    };
27
28  class Table {
29    std::vector<IChart*> v;
30    int data;
31    public:
32      void attach(IChart* p) { v.push_back(p); }
33      void detach(IChart* p) { }
34      void notify(int data) {
35        for(auto p : v) {
36          p->update(data);
37        }
38      }
39
40      void edit(){
41        while(1){
42          std::cout << "data >> ";
43          std::cin >> data;
44          notify(data);
45        }
46      }
47    };
48
49  int main() {
50    Table t;
51
52    BarChart bc; t.attach(&bc);
53    PieChart pc; t.attach(&pc);
54
55    t.edit();
56  }

```

- 1) 모든 차트의 공통의 인터페이스(IChart)가 필요하다
- 2) 테이블에는 다양한 종류의 차트를 보관할 수 있어야 한다(`std::vector<IChart*>`)
- 3) 테이블에는 차트를 등록, 취소하는 함수가 있어야 한다(`attach, detach`)
- 4) 테이블에 있는 데이터에 변화가 생기면 등록된 모든 차트에 통보해야 한다(`notify`)

데이터의 형태가 변경되면 데이터를 편집하는 방법도 수정되어야 한다. 하지만 옵저버 패턴의 기본 로직을 제공하는 `attach, detach, notify는 변경되지 않아야 한다.`

```

1  class Subject {
2    std::vector<IChart*> v;
3    public:
4      virtual ~Subject() {}
5

```

```

6     void attach(IChart* p) { v.push_back(p); }
7     void detach(IChart* p) { }
8     void notify(int data) {
9         for(auto p : v) {
10            p->update(data);
11        }
12    }
13 };
14
15 class Table : public Subject {
16     ...
17 };

```

- 변경되지 않는 코드(옵저버 패턴 기본 로직)을 제공하는 기반 클래스를 제공한다
- **Subject** 클래스 : 관찰 대상의 기반 클래스 (옵저버 패턴의 기본 로직 제공)

4.4.2 Iterator

행위 패턴(behavior pattern)의 한 종류인 반복자 패턴(iterator pattern)은 복합 객체 요소들의 "내부 표현 방식을 공개하지 않고도 순차적으로 접근"할 수 있는 방법을 제공하는 패턴을 말한다.

두 가지 예제 코드에 대해 살펴보자

1. 인터페이스 기반의 설계 : 전통적인 디자인 패턴에서 설명되는 방식 (java, c#, swift, python)
2. STL 설계 : 성능 향상을 위한 기법 (c++)

인터페이스 기반의 설계를 먼저 알아보자

```

1 template<typename T> struct IEnumarator {
2     virtual T& getObject() = 0;
3     virtual bool moveNext() = 0;
4
5     virtual ~IEnumarator() {}
6 };
7
8 template<typename T> struct IEnumarable {
9     virtual IEnumarator<T>* GetEnumerator() = 0;
10
11    virtual ~IEnumarable() {}
12 };
13
14 int main() {
15     slist<int> s = {1,2,3,4,5};
16     vector<int> v = {1,2,3,4,5};
17
18     // 모든 컨테이너에서는 반복자를 꺼낼 수 있어야 한다
19     list_enumerator p1 = s.GetEnumerator();
20     vector_enumerator p2 = v.GetEnumerator();
21
22     // 모든 반복자는 사용법이 동일해야 한다
23     p1.moveNext();
24     p2.moveNext();
25
26     int n1 = p1.getObject();
27     int n2 = p2.getObject();
28 }

```

- **slist** : single linked list
- 본 예제는 c#과 유사한 이름을 사용한다
- c#에서는 반복자를 열거자(enumarator)라고도 부름
- 모든 반복자는 사용법이 동일해야 하므로 반복자의 인터페이스가 필요하다(**IEnumarator<T>**)
- 모든 컨테이너는 반복자를 꺼낼 수 있어야 하므로 반복자를 가진 컨테이너도 인터페이스가 필요하다(**IEnumarable<T>**)

```

1 template<typename T> struct Node {

```

```

2     T data;
3     Node* next;
4     Node(const T& d, Node* n) : data(d), next(n) {}
5   };
6
7   template<typename T>
8   class slist_enumerator : public IEnumarator<T> {
9     Node<T>* current;
10    public:
11      slist_enumerator(Node<T>* p = nullptr) : current(p) {}
12
13      T& get0bject() override { return current->data; }
14
15      bool moveNext() override {
16        current = current->next;
17        return current != nullptr;
18      }
19   };
20
21
22   template<typename T> class slist : public IEnumberable<T> {
23     Node<T>* head = nullptr;
24    public:
25      void push_front(const T& a) { head = new Node<T>(a, head); }
26
27      IEnumarator<T>* GetEnumerator() override {
28        return new slist_enumerator<T>(head);
29      }
30   };
31
32   int main() {
33     slist<int> s;
34     s.push_front(10);
35     s.push_front(20);
36     s.push_front(30);
37     s.push_front(40);
38     s.push_front(50);
39
40     IEnumarator<int>* p = s.GetEnumerator();
41
42     std::cout << p->get0bject() << std::endl; // 50
43     p->moveNext();
44     std::cout << p->get0bject() << std::endl; // 40
45   }

```

- 반복자 구현의 핵심 원리 : 첫번째 요소를 가리키는 포인터를 보관하고 있다가 약속된 방식으로 다음으로 이동하고 요소에 접근하는 것
- 단점1: `s.GetEnumerator()`는 동적 할당된 반복자 객체를 반환한다. C++에서는 반드시 사용한 후 `delete`해야 한다. 라이브러리가 할당하고 사용자가 `delete`하는 코드는 좋은 디자인은 아니다
- 단점2: `moveNext()`, `get0bject()`가 가상함수이다. 수천, 수만개의 요소를 열거한다면 오버헤드가 크다
- 단점3: 모든 컨테이너가 동일한 방법을 사용하지 않는다. 배열의 요소를 접근할 때는 배열의 시작주소를 보관했다가 ++로 이동하게 된다

STL 방식의 반복자 패턴 구현 예제를 살펴보자

```

1   template<typename T>
2   class slist_iterator {
3     Node<T>* current;
4    public:
5      slist_iterator(Node<T>* p = nullptr) : current(p) {}
6
7      inline T& operator*() { return current->data; }
8
9      inline slist_iterator& operator++() {
10        current = current->next;

```

```

11     return *this;
12 }
13
14 // 편의성을 위한 ==, != 연산자 재정의
15 inline bool operator==(const slist_iterator& other) const {
16     return current == other.current;
17 }
18 inline bool operator!=(const slist_iterator& other) const {
19     return current != other.current;
20 }
21 };
22
23 template<typename T> class slist {
24     Node<T>* head = nullptr;
25 public:
26     void push_front(const T& a) { head = new Node<T>(a, head); }
27
28     inline slist_iterator<T> begin() {
29         return slist_iterator<T>(head);
30     }
31
32     inline slist_iterator<T> end() {
33         return slist_iterator<T>(nullptr);
34     }
35 };
36
37 int main() {
38     slist<int> s;
39
40     s.push_front(10);
41     s.push_front(20);
42     s.push_front(30);
43     s.push_front(40);
44     s.push_front(50);
45
46     auto first = s.begin();
47     auto last = s.end();
48
49     while(first != last) {
50         std::cout << *first << std::endl;
51         ++first;
52     }
53 }
```

- 가상함수가 아닌 인라인 치환하여 속도를 높인다 (IEnumerator, IEnumerableable 제거)
- moveNext() 대신 ++ 연산자 재정의, getObject() 대신 * 연산자 재정의
- raw pointer와 동일한 방법으로 사용 가능하도록
- 편의성을 위해 ==, != 연산자도 재정의
- 반복자를 꺼내기 위해 begin(), end() 함수 제공 (GetEnumerator 제거)
- 기본적인 개념은 동일하지만 c++만의 특징을 이용하였다

4.4.3 Visitor

아파트에 여러 세대가 살고 있다고 가정하자. 방문자 패턴에서 방문자는 가스 검침원 같은 사람을 말한다. 가스 검침원은 모든 세대를 방문하면서 가스를 점하는 역할을 한다.

행위 패턴(behavior) 패턴의 한 종류인 방문자 패턴(visitor pattern)은 "요소(각각의 세대)에 대한 연산을 정의"하는 객체를 만들어서 객체(아파트) 내부의 모든 요소에 연산을 수행하는 패턴을 말한다. 어떻게 모든 요소를 방문하게 만들 것인가가 방문자 패턴의 핵심이다.

방문자 패턴에 대한 두 개의 예제를 살펴보자. 첫번째 예제는 좋은 예제는 아니지만 간단하고 이해하기 쉬운 반면에 두번째 예제는 약간 복잡하지만 방문자 패턴을 알기 위한 좋은 예제이다.

첫번째 방법을 먼저 살펴보자

```

1 int main() {
2     std::list<int> s = {1,2,3,4,5,6,7,8,9,10};
```

```

3
4 // #1 직접 연산
5 for(auto &e : s) {
6     e *= 2;
7 }
8
9 // #2 방문자 패턴 사용
10 TwiceVisitor<int> tv; // 요소 한개의 값을 2배로 하는 방문자
11 s.accept(&tv);
12
13 ShowVisitor<int> sv; // 요소 한개를 출력하는 방문자
14 s.accept(&sv);
15 }
```

- 컨테이너에 담긴 모든 요소를 2배로 하고 싶다면?
- 방법1: `for`문으로 모든 요소에 연산을 수행한다
- 방법2: 방문자 패턴을 사용한다
- 규칙1: `s.accept()` 다양한 종류의 방문자 객체를 받을 수 있어야 한다([방문자 인터페이스가 필요하다](#), `IVisitor`)
- 규칙2: 방문의 대상(list, vector 등)은 `accept()`가 있어야 한다([인터페이스가 필요하다](#), `IAcceptor`)

```

1 template<typename T> struct IVisitor {
2     virtual void visit(T& e) = 0;
3     virtual ~IVisitor() {}
4 };
5
6 template<typename T>
7 class TwiceVisitor : public IVisitor<T> {
8     public:
9         void visit(T& e) override { e *= 2; }
10    };
11
12 template<typename T>
13 class ShowVisitor : public IVisitor<T> {
14     public:
15         void visit(T& e) override { std::cout << e << ", "; }
16    };
17
18 template<typename T> struct IAcceptor {
19     virtual void accept(IVisitor<T>* visitor) = 0;
20     virtual ~IAcceptor() {}
21 };
22
23 template<typename T>
24 class MyList : public std::list<T>, public IAcceptor<T> {
25     public:
26         using std::list<T>::list;
27
28         void accept(IVisitor<T>* visitor) override {
29             for(auto &e : *this) {
30                 visitor->visit(e);
31             }
32         }
33    };
34
35 int main() {
36     MyList s = {1,2,3,4,5,6,7,8,9,10};
37
38     TwiceVisitor<int> tv;
39     s.accept(&tv);
40
41     ShowVisitor<int> sv;
42     s.accept(&sv);
43 }
```

-
- 방문자(visitor): **요소 한 개에 대한 연산을 정의하는 객체**이다.
 - 방문의 대상은 accept() 함수를 제공해야 하는데 `std::list`는 accept()가 없으므로 `std::list`, `IAcceptor`를 상속하는 클래스를 만든다

방문자 패턴에 대한 두번째 예제를 살펴보자

```
1 class BaseMenu {
2     std::string title;
3     public:
4     BaseMenu(const std::string& title) : title(title) {}
5     virtual ~BaseMenu() {}
6
7     std::string get_title() const { return title; }
8
9     void set_title(const std::string& s) { title = s; }
10
11    virtual void command() = 0;
12 };
13
14 class MenuItem : public BaseMenu {
15     int id;
16     public:
17     MenuItem(const std::string& title, int id) : BaseMenu(title) {}
18
19     void command() override {
20         std::cout << get_title() << " menu selected" << std::endl;
21         _getch();
22     }
23 };
24
25 class PopupMenu : public BaseMenu {
26     std::vector<BaseMenu*> v;
27     public:
28     PopupMenu(const std::string& title) : BaseMenu(title) {}
29
30     void add_menu(BaseMenu* p) { v.push_back(p); }
31
32     void command() override {
33         while(1) {
34             system("cls");
35
36             std::size_t sz = v.size();
37             std::size_t i = 0;
38
39             for(MenuItem* p : v) {
40                 std::cout << ++i << ". " << p->get_title() << "\n";
41             }
42
43             std::cout << i + 1 << ". finished\n";
44             std::cout << "Please select menu >> ";
45
46             int cmd;
47             std::cin >> cmd;
48
49             if( cmd == sz + 1 ) break;
50             if( cmd < 1 || cmd > sz + 1 ) continue;
51
52             v[cmd-1]->command();
53         }
54     }
55 };
56
57 int main() {
58     PopupMenu* root = new PopupMenu("ROOT");
59     PopupMenu* pm1 = new PopupMenu("해상도 변경");
60     PopupMenu* pm2 = new PopupMenu("색상 변경");
```

```

61     root->add_menu(pm1);
62     root->add_menu(pm2);
63
64     pm1->add_menu(new MenuItem("HD", 11));
65     pm1->add_menu(new MenuItem("FHD", 12));
66     pm1->add_menu(new MenuItem("UHD", 13));
67
68     pm2->add_menu(new MenuItem("RED", 21));
69     pm2->add_menu(new MenuItem("GREEN", 22));
70     pm2->add_menu(new MenuItem("BLUE", 23));
71
72     root->command();
73 }

```

- 위 코드에 방문자 패턴을 적용해보자
- 노드의 트리 구조를 돌아다니면서 PopupMenu 일 때 타이틀을 변경해보자
- 첫번째 예제: list는 모든 요소의 타입이 int로 동일하였다. 모든 요소가 선형적인 형태로 보관된다
- 두번째 예제: Menu는 **요소의 타입이 다양하다**(MenuItem, PopupMenu). 모든 요소가 **트리 구조로 보관**되어 있다

```

1  struct IMenuVisitor {
2     virtual void visit(MenuItem* mi) = 0;
3     virtual void visit(PopupMenu* pm) = 0;
4
5     virtual ~IMenuVisitor() {}
6 };
7
8 struct IAcceptor {
9     virtual void accept(IMenuVisitor* visitor) = 0;
10    virtual ~IAccept() {}
11 };
12 // -----
13
14 class BaseMenu : public IAcceptor {
15     ...
16 };
17
18 class MenuItem : public BaseMenu {
19     void accept(IMenuVisitor* visitor) {
20         visitor->visit(this);
21     }
22     ...
23 };
24
25 class PopupMenu : public BaseMenu {
26     void accept(IMenuVisitor* visitor) {
27         visitor->visit(this);           // Root 노드에 accept 적용
28
29         for(auto child : v) {
30             child->accept(visitor);   // 핵심! 모든 자식 노드에도 accept 적용
31         }
32     }
33     ...
34 };
35
36 class PopupMenuTitleChangeVisitor : public IMenuVisitor {
37 public:
38     void visit(MenuItem* mi) {} // 메뉴 아이템 방문 시 할 일은 없다
39
40     void visit(PopupMenu* pm) { // 팝업 메뉴 방문 시 타이틀을 변경한다
41         std::string s = "[" + pm->get_title() + "]";
42
43         pm->set_title(s);
44     }
45 };

```

```

46
47     int main() {
48     ...
49     PopupMenuItemChangeVisitor v;
50     root->accept(&v);
51 }
```

- IMenuVisitor는 MenuItem, PopupMenu를 다르게 처리하기 위해 visit 함수 두 개를 갖는다
- 팝업 메뉴일 때 accept에서 자기 자신을 visit한 후 child들의 accept을 호출해서 각각 방문자 함수를 호출한다

객체지향 디자인의 특징

- 새로운 요소(타입)를 추가하는 것은 **쉽다**
- 새로운 오퍼레이션(가상함수)을 추가하는 것은 **어렵다**

Shape와 이를 상속하는 Rect, Circle이 있는 경우 Shape에 속하는 Triangle을 추가하는 것은 쉽다. 하지만 Shape에 move()라는 가상함수를 추가하려면 모든 도형을 변경해야 한다.

만약 가상함수를 추가하지 말고 가상함수가 할 일을 방문자로 만들면?

```

1 ShapeMoveVisitor smv;           // move()를 수행하는 방문자
2 shapeContainer.accept(&smv);
```

방문자 패턴의 특징

- 새로운 요소(타입)를 추가하는 것은 **어렵다**
- 새로운 오퍼레이션(방문자로 작성)을 추가하는 것은 **쉬워진다**

예를 들어 SpecialMenu라는 새로운 요소가 추가되면 이에 대한 visit 함수를 따로 작성해줘야 한다. 따라서 새로운 요소(타입)을 추가하는 것이 상대적으로 어려워진다.

```

1 struct IMenuVisitor {
2     virtual void visit(MenuItem* mi) = 0;
3     virtual void visit(PopupMenu* pm) = 0;
4     virtual void visit(SpecialMenu* sm) = 0;
5     virtual ~IMenuVisitor() {}
6 };
```

4.5 Creating and sharing objects

4.5.1 Singleton

생성 패턴(creational pattern)의 한 종류인 싱글톤 패턴은 클래스의 "인스턴스는 오직 하나임을 보장"하며 어디에서든지 "동일한 방법으로 접근"하는 방법을 제공한다.

싱글톤 패턴의 단점과 비판

- 오직 하나이며 동일한 방법으로 접근한다는 말은 결국은 전역변수와 유사
- 멀티스레드 간 접근 문제
- 객체(함수)간의 결합도 증가. 재사용성 감소

```

1 class Cursor {
2     private:
3     Cursor() {}
4
5     Cursor(const Cusor&) = delete;
6     Cursor& operator=(const Cursor&) = delete; // 복사, 대입 생성자 불가
7     public:
8     static Cursor& get_instance() {
9         static Cursor instance;
10        return instance;
11    }
12 };
```

```

14 int main() {
15     Cursor& c1 = Cursor::get_instance();
16     Cursor& c2 = Cursor::get_instance(); // c1,c2는 동일한 객체이다
17
18     Cursor c3 = c1; // error! 복사, 대입 불가
19 }
```

- 싱글톤을 구현하는 기본 규칙
- 1) 외부에서는 객체를 생성할 수 없어야 한다 (**private** 생성자)
- 2) 한 개의 객체는 만들 수 있어야 한다 (**static** 변수)
- 3) 복사 생성자도 사용할 수 없어야 한다 (복사, 대입 금지 =**delete**)

싱글톤을 구현하는 방법은 여러가지가 있다. 위 예제 코드에서는 **Meyer's singleton 패턴**을 사용하였다(Scott Meyer는 effective C++의 저자).

- 오직 한 개의 객체를 **static** 지역변수로 생성한다
- **지연된 초기화**: 처음 **get_instance()**를 호출할 때 초기화된다. 사용되지 않으면 생성자가 호출되지 않는다
- **멀티스레드 안전**: 멀티스레드 환경에서도 **Cursor**는 오직 한 개만 생성됨을 보장한다

```

1 class Cursur {
2     private:
3     Cursor() {
4         std::cout << "start Cursor()" << std::endl;
5         std::this_thread::sleep_for(3s);
6         std::cout << "start Cursor()" << std::endl;
7     }
8
9     Cursor(const Cusor&) = delete;
10    Cursor& operator=(const Cursor&) = delete;
11
12    public:
13        static Cursor& get_instance(){
14            std::cout << "start get_instance()" << std::endl;
15            static Cursor instance;
16            std::cout << "start get_instance()" << std::endl;
17            return instance;
18        }
19    };
20
21    int main() {
22        std::thread t1(&Cursor::get_instance);
23        std::thread t2(&Cursor::get_instance);
24
25        t1.join();
26        t2.join();
27    }
```

- 멀티스레드 환경에서 t1, t2가 **get_instance()**를 호출해도 **static** 변수인 **instance**는 thread-safe하다.
- 싱글톤을 만드는 또 다른 패턴에 대해 살펴보자

```

1 class Cursor {
2     private:
3     Cursor() {}
4     Cursor(const Cursor&) = delete;
5     Cursor& operator=(const Curosrt&) = delete;
6
7     static std::mutex m;
8     static Cursor* instance;
9     public:
10        static Cursor& get_instance() {
11            std::lock_guard<std::mutex> g(m);
12            if( instance == nullptr ) {
13                instance = new Cursor;
14            }
15        }
```

```

15     return *instance;
16 }
17 };
18
19 Cursor* Cursor::instance = nullptr;
20 std::mutex Cursor::m;
21
22 int main() {
23     Cursor& c1 = Cursor::get_instance();
24     Cursor& c2 = Cursor::get_instance();
25 }
```

- 유일한 객체를 힙에 생성하는 모델 (이전 예제는 `static` 지역변수)
- 멀티스레드에 안전하지 않으므로 동기화가 필요하다
- `get_instance()`를 100번 호출하면 최초 호출시 객체를 생성하고 나머지 99번은 생성된 객체를 반환만 한다
- 그런데 `m.lock()`, `m.unlock()`을 계속 수행하게 된다

```

1 static Cursor& get_instance() {
2     if( instance == nullptr ) {
3         m.lock();
4         if( instance == nullptr ) {
5             instance = new Cursor;
6         }
7         m.unlock();
8     }
9     return *instance;
10 }
11 };
```

- **DCLP**(Double Check Locking Pattern): `lock()`을 위해 `if`문을 2번 사용하는 모델
- C++ 진영에서는 버그이므로 사용하면 안됨
- `instance = new Cursor` 코드에서 컴파일러가 성능 최적화를 위해 코드를 변경하는데 `instance`가 더 이상 `nullptr`이 되지 않기 때문에 다른 스레드에서 버그가 발생하게 된다

싱글톤 코드를 자동 생성하는 방법에 대해 살펴보자

```

1 #define MAKE_SINGLETON(classname) \
2 private: \
3     classname() {} \
4     classname(classname&) = delete; \
5     void operator=(classname&) = delete; \
6 public: \
7     static classname& getInstance() { \
8         static classname instance; \
9         return instance; \
10    } \
11 private: \
12 //----- \
13 class Cursor { \
14     MAKE_SINGLETON(Cursor) \
15 };
```

- `#define` 매크로를 사용하여 자동 생성할 수 있다

```

1 template<typename T>          // CRTP
2 class Singleton {
3     private:
4     Singleton() {}
5     Singleton(const Singleton&) = delete;
6     Singleton& operator=(const Singleton&) = delete;
7 }
```

```

8   static std::mutex m;
9   static T* instance;
10  public:
11  static T& get_instance() {
12    std::lock_guard<std::mutex> g(m);
13    if( instance == nullptr ) {
14      instance = new T;
15    }
16    return *instance;
17  }
18 };
19
20 template<typename T> T* Singleton<T>::instance = nullptr;
21 template<typename T> std::mutex Singleton<T>::m;
22
23 class Mouse : public Singleton< Mouse > {
24 ...
25 };
26
27 int main() {
28   Mouse& c1 = Mouse::get_instance();
29 }
```

- 클래스 상속을 이용해서도 자동 생성할 수 있다
- **CRT**P(Curiously Recurring Template Pattern) : 기반 클래스에서 미래에 만들어질 파생 클래스의 이름을 사용할 수 있게 하는 기술
- 기반 클래스를 템플릿으로 만들고 파생 클래스에서 자신의 이름을 기반 클래스의 템플릿 인자로 전달해주는 기술

4.5.2 Factory

전통적인 디자인 패턴 분류에는 factory라는 패턴은 없고 abstract factory 패턴이 존재한다. 하지만 관례적으로 factory라고 불리는 기술이 많이 사용되고 있고 abstract factory 패턴을 이해하려면 먼저 factory 자체를 이해해야 한다.

```

1 class Shape {
2   public:
3     virtual void draw() = 0;
4     virtual ~Shape() {}
5 };
6
7 class Rect : public Shape {
8   public:
9     void draw() override { std::cout << "draw Rect" << std::endl; }
10 };
11
12 class Circle : public Shape {
13   public:
14     void draw() override { std::cout << "draw Rect" << std::endl; }
15 };
16
17 int main() {
18   std::vector<Shape*> v;
19
20   while(1) {
21     int cmd;
22     std::cin >> cmd;
23
24     if(cmd == 1) v.push_back(new Rect);
25     else if(cmd == 2) v.push_back(new Circle);
26     else if(cmd == 9) {
27       for(auto s : v)
28         s->draw();
29     }
30   }
31 }
```

-
- 새로운 도형이 추가되어도 "도형을 생성하는 코드가 변경되지 않게" 할 수 있을까?
 - 팩토리 패턴을 사용해서 도형을 생성하면 된다
-

```
1 class ShapeFactory {
2     MAKE_SINGLETON(ShapeFactory)
3     public:
4     Shape* create(int type) {
5         Shape* p = nullptr;
6         switch(type) {
7             case 1: p = new Rect; break;
8             case 1: p = new Circle; break;
9         }
10        return p;
11    }
12 };
13
14 int main() {
15     std::vector<Shape*> v;
16
17     ShapeFactory& factory = ShapeFactory::get_instance();
18
19     if(cmd > 0 && cmd < 8) // 1 - 7을 도형 번호로 예약
20     {
21         Shape* s = factory.create(cmd);
22         if(s)
23             v.push_back(s);
24     }
25     else if(cmd == 9) {
26         for(auto s : v)
27             s->draw();
28     }
29 }
```

- 장점1: 객체를 생성하는 장소가 한 곳에 집중된다
- 장점2: 객체의 생성 과정을 관리할 수 있고 새로운 타입이 추가되어도 한 곳(팩토리)만 변경된다

팩토리 패턴을 좀 더 발전시켜보자.

```
1 class Shape {
2     public:
3     virtual void draw() = 0;
4     virtual ~Shape() {}
5 };
6
7 class Rect : public Shape {
8     public:
9     void draw() override { std::cout << "draw Rect" << std::endl; }
10
11    static Shape* create() { return new Rect; }
12 };
13
14 class Circle : public Shape {
15     public:
16     void draw() override { std::cout << "draw Rect" << std::endl; }
17
18    static Shape* create() { return new Circle; }
19 };
```

- static 멤버 함수를 사용하여 새로운 기법으로 객체를 생성할 수 있다
- 기존: Rect* r1 = new Rect;
- 새로운: Rect* r2 = Rect::create();
- C++에서는 클래스 이름을 자료구조에 보관할 수 없다.

-
- `v.push_back("Rect")`는 문자열을 보관하는 것이지 클래스 정보를 보관하는 것이 아니다
 - 하지만 `Rect` 객체를 만드는 생성함수를 자료구조에 보관할 수는 있다

```
1 class ShapeFactory {
2     MAKE_SINGLETON(ShapeFactory)
3
4     using F = Shape*();           // 함수 포인터
5
6     std::map<int, F> create_map; // 도형의 번호, 생성 함수 등록
7
8     public:
9     void register_shape(int key, F create_function) {
10         create_map[key] = create_function;
11     }
12
13     Shape* create(int type) {
14         Shape* p = nullptr;
15         if(create_map[type] != nullptr) {
16             p = create_map[type](); // 도형 생성
17         }
18         return p;
19     }
20 };
21
22 int main() {
23     std::vector<Shape*> v;
24
25     ShapeFactory& factory = ShapeFactory::get_instance();
26
27     // 공장에 제품 등록
28     factory.register_shape(1, &Rect::create);
29     factory.register_shape(2, &Circle::create);
30
31     if(cmd > 0 && cmd < 8)
32     {
33         Shape* s = factory.create(cmd);
34         if(s)
35             v.push_back(s);
36     }
37     else if(cmd == 9) {
38         for(auto s : v)
39             s->draw();
40     }
41 }
```

-
- 새로운 제품이 추가되는 경우 `cmd` 관련 코드도 변경되지 않고 팩토리 패턴 코드도 변경되지 않는다. 오직 [공장에 등록하는 `register_shape\(\)` 함수만 변경](#)하면 된다.

공장에 제품을 자동 등록하는 방법에 대해 알아보자

```
1 class Shape {
2     ...
3 };
4
5 class ShapeFactory {
6     ...
7 };
8
9 class RegisterFactory {
10     public:
11     RegisterFactory(int type, Shape* (*f)()) {
12         ShapeFactory::get_instance().register_shape(type, f);
13     }
14 };
15
```

```

16 class Rect : public Shape {
17     ...
18     static RegisterFactory rf;
19 };
20
21 class Circle : public Shape {
22     ...
23     static RegisterFactory rf;
24 };
25
26 RegisterFactory Rect::rf(1, &Rect::create);
27 RegisterFactory Circle::rf(2, &Circle::create);           // 컴파일 타임에 이미 공장에 등록된다
28
29 int main() {
30     std::vector<Shape*> v;
31
32     ShapeFactory& factory = ShapeFactory::get_instance();
33
34     if(cmd > 0 && cmd < 8) {
35         Shape* s = factory.create(cmd);
36         if(s)
37             v.push_back(s);
38     }
39     else if(cmd == 9) {
40         for(auto s : v)
41             s->draw();
42     }
43 }
```

- 이제 main에서도 팩토리에서도 코드를 수정할 필요가 없게 되었다
- 클래스를 만드는 사람이 직접 RegisterFactory를 사용하여 공장에 등록해야 한다

모든 클래스는 생성할 때 create 함수와 RegisterFactory를 반드시 선언해야 한다. 모든 클래스가 해당 코드를 공유하기 때문에 매크로로 묶어주는 것이 가능하다

```

1 #define REGISTER(classname) \
2     static Shape* create() { return new classname; } \
3     static RegisterFactory rf;
4
5 #define REGISTER_IMPL(type, classname) \
6     RegisterFactory classname::rf(type, &classname::create);
7
8 class Rect : public Shape {
9     ...
10    REGISTER(Rect)
11 };
12
13 class Circle : public Shape {
14     ...
15    REGISTER(Circle)
16 };
17
18 REGISTER_IMPL(1, Rect)
19 REGISTER_IMPL(2, Circle)
```

4.5.3 Prototype

생성 패턴(creational pattern) 중 하나인 프로토타입 패턴은 견본 인스턴스를 사용하여 생성할 객체의 종류를 명시하고 이렇게 만들어진 "견본을 복사하여 새로운 객체를 생성"하는 패턴을 말한다.

```

1 int main() {
2     std::vector<Shape*> v;
3     ShapeFactory& factory = ShapeFactory::get_instance();
4
5     factory.register_shape(1, &Rect::create);
```

```

6   factory.register_shape(2, &Circle::create);
7
8   while(1) {
9     int cmd;
10    std::cin >> cmd;
11
12    if(cmd > 0 && cmd < 8) {
13      Shape* s = factory.create(cmd);
14      if(s)
15        v.push_back(s);
16    }
17  }
18 }
```

- register_shape() 함수를 보면 공장에 Rect, Circle 등의 "클래스(정확히는 생성함수)"를 등록하고 있다
- 클래스가 아닌 자주 사용하는 객체를 등록하는 방법은 없을까?

```

1 class Shape {
2   public:
3   ...
4   virtual Shape* clone() = 0;
5 };
6
7 class Rect : public Shape {
8   public:
9   ...
10  Shape* clone() override { return new Rect(*this); }
11 };
12
13 class Circle : public Shape {
14   public:
15   ...
16  Shape* clone() override { return new Circle(*this); }
17 };
18
19 class ShapeFactory {
20   MAKE_SINGLETON(ShapeFactory)
21
22   // F -> Shape*
23   std::map<int, Shape*> prototype_map;
24   public:
25   // register_shape -> register_sample
26   void register_sample(int key, Shape* sample) {
27     prototype_map[key] = sample;
28   }
29
30   Shape* create(int type) {
31     Shape* p = nullptr;
32
33     if(prototype_map[type] != nullptr) {
34       p = prototype_map[type]->clone();
35     }
36     return p;
37   }
38 };
39
40 int main() {
41   std::vector<Shape*> v;
42   ShapeFactory& factory = ShapeFactory::get_instance();
43
44   Rect* red_rect = new Rect;
45   Rect* blue_rect = new Rect;
46   Circle* blue_circle = new Circle;
47
48   // 공장에 생성함수가 아닌 객체를 등록한다
```

```

49 factory.register_sample(1, red_rect);
50 factory.register_sample(2, blue_rect);
51 factory.register_sample(3, blue_circle);
52
53 while(1) {
54     int cmd;
55     std::cin >> cmd;
56
57     if(cmd > 0 && cmd < 8) {
58         Shape* s = factory.create(cmd);
59         if(s)
60             v.push_back(s);
61     }
62 }
63 }
```

- 프로토타입은 아무것도 없는 상태에서 생성함수를 통해 객체를 만드는 것이 아니라 **자주 사용하는 객체들을 등록하였다가 견본 객체를 복사하여 새로운 객체를 만드는 방법**을 말한다.

객체를 생성하는 방법은 다음과 같다.

Rect r;	객체의 수명이 정해져 있다 사용자가 원할 때 파괴할 수 없다
r = new Rect;	가장 자유로운 방법 자유롭게 생성하고 자유롭게 파괴
r = Rect::create();	객체의 생성을 한 곳에서. 다양한 제약을 사용할 수 있다. 오직 한 개만 만들 수 있게 하려면 (singleton) 속성이 동일하면 공유 (flyweight) 생성함수 주소를 자료구조에 보관할 수도 있다
r = factory.create();	공장을 통한 객체의 생성 (factory)
r = sample.clone();	기존 객체에 복사를 통한 새로운 객체 생성 (prototype)

4.5.4 Abstract factory

생성 패턴(creational pattern) 중 하나인 추상 팩토리 패턴은 상세화된 서브 클래스를 정의하지 않고도 서로 관련성이 있거나 독립적인 "여러 객체군을 생성하기 위한 인터페이스를 제공"하는 패턴이다.

```

1 struct IButton {
2     virtual void draw() = 0;
3     virtual ~IButton() {}
4 };
5
6 struct IEdit {
7     virtual void draw() = 0;
8     virtual ~IEdit() {}
9 };
10
11 struct RichButton : public IButton {
12     void draw() { std::cout << "draw RichButton" << std::endl; }
13 };
14
15 struct RichEdit : public IEdit {
16     void draw() { std::cout << "draw RichEdit" << std::endl; }
17 };
18
19 struct SimpleButton : public IButton {
20     void draw() { std::cout << "draw SimpleButton" << std::endl; }
21 };
22
23 struct SimpleEdit : public IEdit {
24     void draw() { std::cout << "draw SimpleEdit" << std::endl; }
25 };
```

```

26
27 int main() {
28     IButton* btn;
29
30     if(strcmp(argv[1], "-style:rich") == 0)
31         btn = new RichButton;
32     else
33         btn = new SimpleButton;
34
35     btn->draw();
36 }
```

- GUI 라이브러리 안에 2가지 종류의 컨트롤 클래스를 제공한다고 가정하자
- RichButton, Edit, ... : 예쁘고 화려하지만 메모리 사용량이 많고 성능이 좋지 않음. 고성능 시스템에 적합
- SimpleButton, Edit, ... : 단순하고 예쁘지 않지만 메모리 사용량이 적고 빠르게 동작
- 사용자가 옵션으로 종류를 선택 가능하게 하고자 한다

만약 컨트롤 클래스가 30개 이상이 넘어간다고 하면 일일히 컨트롤 클래스를 사용하여 생성하기 어렵다. (Rich에 30개, Simple에 30개, Fancy에 30개를 매 번 따로 생성해야 함)

```

1 // 추상 팩토리 패턴
2 struct IControlFactory {
3     IButton* create_button() = 0;
4     IEdit* create_edit() = 0;
5     virtual ~IControlFactory() {}
6 };
7
8 class RichControlFactory : public IControlFactory {
9     public:
10    IButton* create_button() { return new RichButton; }
11    IEdit* create_edit() { return new RichEdit; }
12 };
13
14 class SimpleControlFactory : public IControlFactory{
15     public:
16    IButton* create_button() { return new SimpleButton; }
17    IEdit* create_edit() { return new SimpleEdit; }
18 };
19
20 int main(){
21     IControlFactory* factory;
22
23     if(strcmp(argv[1], "-style:rich") == 0)
24         factory = new RichControlFactory;
25     else
26         factory = new SimpleControlFactory;
27
28     IButton* btn = factory->create_button();
29     btn->draw();
30 }
```

4.5.5 Flyweight

구조 패턴(structural pattern)의 한 종류인 플라이웨이트(flyweight) 패턴은 속성이 동일한 객체를 공유하게 하는 패턴을 말한다.

```

1 class Image{
2     std::string image_url;
3     public:
4     Image(const std::string& url) : image_url(url) {
5         std::cout << url << " Downloading...\n";
6     }
7     void draw() {
8         std::cout << "draw " << image_url << '\n';
```

```

9     }
10 };
11
12 int main() {
13     Image* img1 = new Image("www.image.com/a.png");
14     img1->draw();
15
16     Image* img2 = new Image("www.image.com/a.png"); // 동일한 그림 다운로드
17     img2->draw();
18 }
```

- `Image` 클래스 : 그림을 관리하는 클래스. 파일 또는 인터넷에서 다운로드하는 기능 지원
- `img1`에서 다운로드한 그림을 `img2`에서 또 다운로드하고 있다 (비효율)
- 플라이웨이트 패턴 : 속성이 동일하면 공유하게 하자. 동일한 그림을 관리하는 객체를 2개 만들 필요가 없다

```

1 class Image{
2     std::string image_url;
3
4     // 생성자 private 이동
5     Image(const std::string& url) : image_url(url) {
6         std::cout << url << " Downloading...\n";
7     }
8
9     public:
10    void draw() {
11        std::cout << "draw " << image_url << '\n';
12    }
13
14    static std::map<std::string, Image*> image_map;
15
16    static Image* create(const std::string& url) {
17        Image* img;
18        auto ret = image_map.find(url);
19
20        // 이미 다운로드한 그림이 아닌 경우에만 새로 다운로드
21        if(ret == image_map.end()) {
22            img = new Image(url);
23            image_map[url] = img;
24        }
25        return image_map[url];
26    }
27
28 int main() {
29     Image* img1 = Image::create("www.image.com/a.png");
30     img1->draw();
31
32     Image* img2 = Image::create("www.image.com/a.png");
33     img2->draw();
34 }
```

- `static` 멤버 함수를 사용하여 플라이웨이트 패턴 구현

위 코드에서 `Image` 클래스는 이미지를 그리는 `draw` 기능과 자신을 생성하는 `create` 기능을 둘 다 수행한다. 하나의 클래스는 하나의 기능을 가지는 것이 이상적이므로 생성 기능을 분리할 수 있다

```

1 class Image {
2     std::string image_url;
3
4     Image(const std::string& url) : image_url(url) {
5         std::cout << url << " Downloading...\n";
6     }
7
8     public:
9     void draw() {
10        std::cout << "draw " << image_url << '\n';
```

```

10     }
11
12     friend class ImageFactory;
13 }
14
15 class ImageFactory {
16     std::map<std::string, Image*> image_map;
17 public:
18     Image* create(const std::string& url) {
19         Image* img;
20         auto ret = image_map.find(url);
21
22         if(ret == image_map.end()) {
23             img = new Image(url);
24             image_map[url] = img;
25         }
26         return image_map[url];
27     }
28 };
29
30 int main() {
31     ImageFactory factory;
32
33     Image* img1 = factory.create("www.image.com/a.png");
34     img1->draw();
35
36     Image* img2 = factory.create("www.image.com/a.png");
37     img2->draw();
38 }
```

- Image 클래스의 생성자가 `private`인데 ImageFactory에서 생성해야 하므로 `friend` 문법을 사용한다

MS Word에서 각각의 글자는 다양한 속성을 개별적으로 변경할 수 있다.(ABCDEFGHIJ). 이는 각각의 글자들이 개별적인 클래스인 것을 의미하며 이는 다음과 같다.

```

1 class Char {
2     char value;
3     int font_size;
4     int font_weight;
5     COLOR font_color;
6     std::string font_familiy;
7     ...
8 };
```

대부분의 경우 폰트 관련 속성은 공유되는 경우가 많은데 모든 글자마다 속성들이 개별적으로 존재한다면 메모리 사용량이 매우 많을 것이다. 공유되는 속성을 클래스로 분리하고 이를 모든 글자들이 공유하도록 하는 플라이웨이트 패턴을 사용하여 메모리를 효율적으로 관리한다.

```

1 class Char{
2     char value;
3     Font* font;
4 };
5
6 class Font {
7     int font_size;
8     int font_weight;
9     COLOR font_color;
10    std::string font_familiy;
11    ...
12 };
13
14 class FontFactory {
15    ...
16 };
```

4.5.6 Builder

```
1 // 입학지원서
2 using Application = string; // class Application {}를 단순화하여 string으로 간주
3
4 // 지원서 만드는 클래스
5 class Director {
6     string name = "HONG";
7     string phone = "010-111-1111";
8     string address = "SEOUL KANGNAMGU";
9     public:
10    Application construct() {
11        Application app;
12        app += name + "\n";
13        app += phone + "\n";
14        app += address + "\n";
15        return app;
16    };
17 }
18
19 int main() {
20     Director d;
21
22     Application app = d.construct();
23
24     std::cout << app << std::endl;
25 }
```

- 지원서 객체는 다양한 형태로 생성될 수 있어야 한다
- 지원서에는 이름, 전화번호, 주소의 순서로 표기되어야 한다
- 지원서는 상황에 따라 raw, html, xml 방식으로 만들 수 있어야 한다

```
1 // 입학지원서
2 using Application = string;
3
4 // 지원서 만드는 클래스
5 class Director {
6     string name = "HONG";
7     string phone = "010-111-1111";
8     string address = "SEOUL KANGNAMGU";
9     public:
10    Application construct() {
11        Application app;
12        app += name + "\n";
13        app += phone + "\n";
14        app += address + "\n";
15        return app;
16    };
17
18    Application XMLconstruct() {
19        Application app;
20        app += "<NAME>" + name + "</NAME>\n";
21        app += "<PHONE>" + phone + "</PHONE>\n";
22        app += "<ADDRESS>" + address + "</ADDRESS>\n";
23        return app;
24    };
25 }
26
27 int main() {
28     Director d;
29
30     Application app = d.XMLconstruct();
31
32     std::cout << app << std::endl;
```

- xml 방식을 지원하기 위해 XMLconstruct() 함수를 만들었다
- 만약 지원서의 표기 방식이 바뀌어 주소(address)를 표기하지 말아야 하는 경우에는 construct(), XMLconstruct() 두 함수를 모두 수정해야 한다. 포맷이 다양할 수록 모든 함수를 수정해야 한다
- **변하는 것은 분리되어야 한다** (디자인 패턴 원칙)
- 방법1: 변하는 것을 가상함수로
- 방법2: 변하는 것을 클래스로

변하는 부분을 다른 클래스로 바꾸는 코드를 살펴보자

```

1  using Application = string;
2
3  struct IBuilder {
4      virtual ~IBuilder() {}
5      virtual void makeName(string name) = 0;
6      virtual void makePhone(string phone) = 0;
7      virtual void makeAddress(string addr) = 0;
8
9      virtual Application getResult() = 0;
10 };
11
12 class XMLBuilder : public IBuilder {
13     Application app;
14     public:
15     void makeName(string name) override { app += "<NAME>" + name + "</NAME>\n"; }
16     void makePhone(string phone) override { app += "<PHONE>" + phone + "</PHONE>\n"; }
17     void makeAddress(string address) override { app += "<ADDRESS>" + address + "</ADDRESS>\n"; }
18
19     Application getResult() override { return app; }
20 };
21
22 class TextBuilder : public IBuilder {
23     Application app;
24     public:
25     void makeName(string name) override { app += name + "\n"; }
26     void makePhone(string phone) override { app += phone + "\n"; }
27     void makeAddress(string address) override { app += address + "\n"; }
28
29     Application getResult() override { return app; }
30 };
31
32 class Director {
33     string name = "HONG";
34     string phone = "010-111-1111";
35     string address = "SEOUL KANGNAMGU";
36
37     IBuilder* pBuilder;
38     public:
39     void setBuilder(IBuilder* p) { pBuilder = p; }
40
41     Application construct() {
42         pBuilder->makeName(name);
43         pBuilder->makePhone(phone);
44         pBuilder->makeAddress(address);
45
46         return pBuilder->getResult();
47     };
48 };
49
50 int main() {
51     Director d;
52
53     XMLBuilder xb;
54     d.setBuilder(&xb);

```

```

55
56     Application app = d.construct();
57     std::cout << app << std::endl;
58
59     TextBuilder tb;
60     d.setBuilder(&tb);
61
62     app = d.construct();
63     std::cout << app << std::endl;
64 }
```

- 빌더 패턴의 의도: 복잡한 객체를 "생성하는 방법과 표현하는 방법을 정의하는 클래스를 별도로 분리"하여 서로 다른 표현이라도 이를 생성할 수 있는 동일한 구축 공정을 제공할 수 있도록 한다
- 전략 패턴, 상태 패턴과 비슷해보일 수 있다. 셋은 클래스 다이어그램도 동일하다
- 전략 패턴: 알고리즘을 바꿀 때 사용하는 패턴
- 상태 패턴: 상태를 바꿀 때 사용하는 패턴
- 빌더 패턴: 객체를 만들 때 사용하는 패턴

4.6 MISC

4.6.1 Memento

행위 패턴(behavior pattern)의 한 종류인 메멘토 패턴은 캡슐화를 위배하지 않고 "객체 내부 상태를 캡슐화해서 저장하고, 나중에 객체가 이 상태로 복구" 가능하게 하는 패턴을 말한다.

```

1 class Graphic{
2     // 윈도우의 그림을 그리는 수많은 함수들을 제공
3 };
4
5 int main() {
6     Graphic g;
7
8     // 그림을 그릴 때 선의 색상이나 두께 등을 변경하고 싶다
9
10    // #1. 함수 인자로 전달
11    g.draw_line(0,0,100,100,red,10);
12    g.draw_line(0,0,200,200,blue,20);
13
14
15    // #2. Graphic 객체의 속성으로 전달
16    g.set_stroke_color(red);
17    g.set_stroke_width(10);
18
19    g.draw_line(0,0,100,100);
20 }
```

- 그림을 그릴 때 선의 색상이나 두께 등을 변경하고 싶다
- 방법1: 함수 인자로 전달
- 방법2: Graphic 객체의 속성으로 지정

두번째 방법을 활용한 예제를 살펴보자

```

1 enum color { red = 1, green = 2, blue = 3 };
2
3 class Graphic {
4     struct Memento {
5         int penWidth;
6         int penColor;
7         Memento(int w, int c) : penWidth(w), penColor(c) {}
8     };
9     std::map<int, Memento*> memento_map;
10
11     int penWidth = 1;
12     int penColor = 0;
```

```

13     int temporary_data;
14
15     public:
16     int save() {
17         static int key = 0;
18         ++key;
19
20         Memento* p = new Memento(penWidth, penColor);
21         memento_map[key] = p;
22
23         return key;
24     }
25     void restore(int token) {
26         penColor = memento_map[token]->penColor;
27         penWidth = memento_map[token]->penWidth;
28     }
29
30     void draw_line(int x1, int y1, int x2, int y2) { }
31     void set_stroke_color(int c) { penColor = c; }
32     void set_stroke_width(int w) { penWidth = w; }
33 };
34
35 int main() {
36     Graphic g;
37
38     g.set_stroke_color(red);
39     g.set_stroke_width(10);
40     g.draw_line(0,0,100,100);
41     g.draw_line(0,0,200,200);
42
43     int token = g.save();           // 메멘토 패턴
44
45     g.set_stroke_color(blue);
46     g.set_stroke_width(20);
47     g.draw_line(0,0,300,300);
48     g.draw_line(0,0,400,400);
49
50     // 처음 그렸던 선과 동일한 속성으로 그리고 싶다
51     g.restore(token);            // 메멘토 패턴
52
53     g.draw_line(10,20,300,300);
54     g.draw_line(10,30,400,400);
55 }
```

- 메멘토 패턴: 객체의 내부 상태를 저장하였다가 복구할 수 있는 패턴

4.6.2 State pattern

게임의 캐릭터는 아이템을 획득함에 따라 동작이 달라진다. 아이템의 상태에 따라 "달리기", "공격하기" 동작을 변경해야 한다.

```

1 class Character {
2     int gold = 0;
3     int item = 0;
4
5     public:
6     void run() {
7         if(item == 1) cout << "run" << endl;
8         else if(item == 2) cout << "fast run" << endl;
9     }
10    void attack() {
11        if(item == 1) cout << "attack" << endl;
12        else if(item == 2) cout << "attack2" << endl;
13    }
14 };
```

```

15 int main() {
16     Character* p = new Character;
17     p->run();
18     p->attack();
19 }
```

- item에 따른 조건 분기가 모든 동작 함수에 필요하다
- 새로운 아이템 추가 시 모든 동작 함수에 분기문이 추가된다

두번째 방법으로 변하는 것을 가상함수로 변경해보자

```

1 class Character {
2     int gold = 0;
3     int item = 0;
4     public:
5     void run() { runImp(); }
6     void attack() { attackImp(); }
7
8     virtual void runImp() {}
9     virtual void attackImp() {}
10 };
11
12 class PowerItemCharacter : public Character {
13     public:
14     void runImp() override { cout << "fast run" << endl; }
15     void attackImp() override { cout << "attack" << endl; }
16 };
17
18 class NormalCharacter : public Character {
19     public:
20     void runImp() override { cout << "run" << endl; }
21     void attackImp() override { cout << "attack" << endl; }
22 };
23
24 int main() {
25     // 일반적인 캐릭터
26     Character* p = new NormalCharacter;
27     p->run();
28     p->attack();
29
30     // 아이템 획득 후 캐릭터
31     p = new PowerItemCharacter;
32     p->run();
33     p->attack();
34 }
```

- PowerItemCharacter는 NormalCharacter와는 상태가 다른 새로운 객체가 된다
- 기존 객체의 동작이 변경된 것이 아닌 **동작이 변경된 새로운 객체가 생성된 것**
- 상속은 객체에 대한 변화가 아닌 **클래스에 의한 변화**이다

세번째 방법으로 변하는 것을 다른 클래스로 분리한다

```

1 struct IState {
2     virtual void run() = 0;
3     virtual void attack() = 0;
4     virtual ~IState() {}
5 };
6
7 class Character {
8     int gold = 0;
9     int item = 0;
10    IState* state;
11    public:
12    void changeState(IState* p) { state = p; }
13 }
```

```

14     void run() { state->run(); }
15     void attack() { state->attack(); }
16 };
17
18 class NormalState : public IState {
19     void run() override { cout << "run" << endl; }
20     void attack() override { cout << "attack" << endl; }
21 };
22
23 class PowerItemState : public IState {
24     void run() override { cout << "fast run" << endl; }
25     void attack() override { cout << "power attack" << endl; }
26 };
27
28 int main() {
29     NormalState ns;
30     PowerItemState ps;
31
32     // 정상 상태
33     Character* p = new Character;
34     p->changeState(&ns);
35     p->run();
36     p->attack();
37
38     // 아이템 획득
39     p->changeState(&ps);
40     p->run();
41     p->attack();
42 }
```

- item에 따른 run(), attack() 함수의 동작을 정의한 클래스를 별도로 제공한다

- **객체의 속성은 유지하지만 동작을 변경할 수 있다**

- 클래스가 아닌 객체에 대한 변화이다

상태 패턴(state pattern)은 객체 자신의 **내부 상태에 따라 동작을 변경**하도록 한다. 객체는 마치 클래스를 바꾸는 것처럼 보인다. 전략 패턴(strategy pattern)은 상태 패턴과 다이어그램도 동일하고 유사해보일 수 있으나 다양한 알고리즘이 존재하면 이들을 캡슐화하여 알고리즘 대체가 가능하도록 하는 점에서 차이점이 있다.

5 Concurrent programming

5.1 Warming up

5.1.1 Concurrent intro

본 섹션에서는 Modern c++에서 지원하는 멀티스레드 관련 기술에 대해 배운다.

thread management	<code>std::thread, std::promise, std::future, std::packaged_task, std::async</code>
thread synchronization	<code>std::mutex</code> 등 6개의 표준 뮤텍스 lock을 다루는 4가지 기술 <code>conditional_variable, semaphore, TLS, call_once</code>
atomic operation	<code>std::atomic, std::memory_order</code>
STL and concurrency	병렬 알고리즘, atomic, smart pointer
c++20	<code>std::semaphore, std::jthread, std::latch, std::barrier, std::atomic_ref</code>
reference	<code>chrono 라이브러리, std::ref</code>

컴파일러는 g++11.0(mingw-w64)를 사용하였으며 빌드하는 방법은 `g++ source.cpp -std=c++20 -lpthread`를 입력하면 된다.

5.1.2 `std::this_thread` namespace

c++11부터 멀티스레드 환경을 지원하면서 `std::this_thread` 네임스페이스를 지원하였다. `std::this_thread` 네임스페이스 안에는 4개의 스레드 관련 함수가 제공된다.

- `std::this_thread::get_id()` : 현재 스레드의 id를 반환
- `std::this_thread::sleep_for()` : 주어진 시간만큼 현재 스레드 재우기
- `std::this_thread::sleep_until()` : 주어진 시간까지 현재 스레드 재우기
- `std::this_thread::yield()` : 다른 스레드를 실행할 수 있도록 힌트 제공

```
1 int main() {
2     std::cout << std::this_thread::get_id() << std::endl;
3
4     std::thread::id tid1 = std::this_thread::get_id();
5     std::thread::id tid2 = std::this_thread::get_id();
6
7     tid1 == tid2;
8     tid1 < tid2;
9
10    std::hash<std::thread::id> h;
11
12    std::cout << h(tid1) << std::endl;
13 }
```

- `std::this_thread::get_id()`: 실행중인 현재 스레드의 id를 반환한다. 반환값이 `int`가 아니라 `std::thread::id`임에 유의한다
- `std::thread::id`: 스레드의 id를 나타내는 구조체. `cout`으로 출력 가능하고 비교 연산이 가능하다. 단, 정수로 변환이 안된다
- `std::hash<std::thread::id>` 함수 객체가 제공되므로 `unordered` 컨테이너에 키 값으로 사용이 가능하다

```
1 int main() {
2     std::this_thread::sleep_for(std::chrono::seconds(3));
3     std::this_thread::sleep_for(3s);
4     std::this_thread::sleep_for(3); // error! chrono 형식이 아님
5
6     std::chrono::time_point tp1 = std::chrono::steady_clock::now();
7     std::this_thread::sleep_until(tp1 + 2000ms); // 현재 시간보다 2초 뒤까지 재운다
8
9     auto tp2 = createDateTime(2021, 4, 11, 12, 39, 00);
10    std::this_thread::sleep_until(tp2); // tp2 시간까지 재운다
11 }
```

두 함수의 구현 코드를 살펴보자

```
1 template<class Rep, class Period>
2 void sleep_for(const std::chrono::duration<Rep, Period>& sleep_duration);
3
4 template<class Clock, class Duration>
5 void sleep_until(const std::chrono::time_point<Clock, Duration>& sleep_time);
```

- 두 함수의 인자가 서로 다른 chrono 타입으로 넘어간다
-

```
1 void mysleep(std::chrono::microseconds us) {
2     auto target = std::chrono::high_resolution_clock::now() + us;
3
4     // 목표 시간이 안됐으면 다른 스레드에 양보한다
5     while(std::chrono::high_resolution_clock::now() < target) {
6         std::this_thread::yield();
7     }
8 }
9
10 int main() {
11     mysleep(1000ms);
12 }
```

- `std::this_thread::yield()` : 현재 스레드의 실행 흐름을 다른 스레드에 양보할 때 사용

5.1.3 Chrono

c++ 표준에서 시간을 다루는 chrono 라이브러리에 대해 살펴보자.

```
1 int main() {
2     std::chrono::hours h(10);
3     std::chrono::minutes m(10);
4     std::chrono::seconds s1(10);
5     std::chrono::milliseconds s2(10);
6     std::chrono::nanoseconds s3(10);
7
8     std::cout << s1.count() << std::endl;
9 }
```

- `<chrono>` 헤더 사용
- `std::chrono` 네임스페이스 사용

chrono 라이브러리의 모든 시간 타입은 `duration<>`의 alias이다.

```
1 using nanoseconds = duration<long long, nano>;
2 using microseconds = duration<long long, micro>;
3 using milliseconds = duration<long long, milli>;
4 using secondsduration<long long>;
5 using minutes = duration<int, ratio<60>>;
6 using hours = duration<int, ratio<3600>>;
```

chrono 전용 유저 정의 리터럴이 존재한다

```
1 using namespace std::literals
2
3 int main() {
4     auto a1 = 10s;
5     auto a2 = 10ms;
6     auto a3 = 10min;
7 }
```

```
1 // 동일한 코드
2 std::chrono::seconds s1(10);
```

```

3 std::chrono::duration<long long> d1(10);
4
5 // 동일한 코드 2
6 std::this_thread::sleep_for( std::chrono::duration<long long>(10) );
7 std::this_thread::sleep_for( std::chrono::seconds(10) );
8 std::this_thread::sleep_for( 10s );                                // 유저 정의 리터럴

```

`std::chrono::time_point`의 개념에 대해 살펴보자. 우선 `duration`과 차이점은 다음과 같다

<code>duration</code>	값과 단위로 구성되어 있음 (30,1) → 30초(seconds) (30,1/1000) → 30밀리초(milliseconds) <code>using seconds = duration<long long, ratio<1,1>;</code> <code>using milliseconds = duration<long long, ratio<1, 1000>;</code>
<code>time_point</code>	기준 시간 + <code>duration</code> 으로 구성되어 있음 epoch time: 1970년 1월 1일 + <code>duration</code>

```

1 int main() {
2     std::chrono::time_point tp1 = std::chrono::system_clock::now();
3
4     // 현재 시간을 1970년 1월 1일을 기준으로 몇시간이 지났는지 알려줌
5     std::chrono::hours h = std::chrono::duration_cast<std::chrono::hours> ( tp1.time_since_epoch() );
6
7     // 시간을 출력
8     std::cout << h.count() << std::endl;
9 }

```

5.2 thread basic

5.2.1 `std::thread`

c++ 표준으로 스레드를 만드는 방법을 살펴보자

```

1 #include <thread>
2
3 void foo() {
4     for(int i=0; i<10; i++) {
5         std::cout << "foo: " << i << std::endl;
6         std::this_thread::sleep_for(100ms);
7     }
8 }
9
10 int main() {
11     std::thread t(&foo);
12     t.join();
13 }

```

- 스레드를 생성하는 방법: `std::thread`의 생성자 인자로 스레드에서 수행할 함수를 전달하면 된다
- `<thread>` 헤더가 필요하다
- `join()`, `detach()` 없이 프로그램이 종료된다면 "terminate called without an active exception"이라는 경고문이 발생한다
- `t.join()` : 스레드의 종료를 대기
- `t.detach()` : 스레드를 떼어낸다(생성된 스레드가 독립적으로 실행)

스레드 함수의 인자에 대해 살펴보자

```

1 void f1() {}
2 void f2(int a, double b) {}
3 void f3(int a, int& b, std::string&& s) { b = 100; }
4
5 int main() {
6     int n = 0;
7     std::string s = "hello";

```

```

8     std::thread t1(&f1);
9     std::thread t2(&f2, 10, 3.4);
10    std::thread t3(&f3, 10, std::ref(n), std::move(s));
11    t1.join();
12    t2.join();
13    t3.join();
14
15    std::cout << s << std::endl; // ""
16    std::cout << n << std::endl; // 100
17
18 }
```

- `std::thread`의 생성자는 "가변인자 템플릿"으로 구현되어 있으므로 생성자에 전달하면 된다
- 스레드 함수의 인자를 참조로 전달할 때는 lvalue reference는 `std::ref()`로 전달하고 rvalue reference는 `std::move()`를 사용한다

스레드와 callable object에 대해 살펴보자. 일반 함수뿐만 아니라 다양한 함수를 스레드로 수행할 수 있다 (일반함수/멤버함수/함수객체/람다표현식 등)

```

1 void foo(int a, double d) {}
2
3 struct Machine {
4     void Run(int a, double d) {}
5 };
6
7 struct Work{
8     void operator()(int a, double b) const {}
9 };
10
11 int main() {
12     Machine m;
13     Work w;
14
15     std::thread t1(&foo, 1, 3.4);
16     std::thread t2(&Machine::Run, &m, 1, 3.4);           // 멤버함수
17     std::thread t3(w, 1, 3.4);                          // 괄호연산자 재정의
18     std::thread t4( [] { std::cout << "lambda" << std::endl; } ); // 람다함수
19
20     t1.join();
21     t2.join();
22     t3.join();
23     t4.join();
24 }
```

스레드 클래스의 다양한 멤버들에 대해 살펴보자

- **멤버 타입**

- `native_handle_type` : `native_handle()` 멤버 함수의 반환타입

- **멤버 클래스**

- `id` : thread id 담는 타입

- **멤버 함수**

- `hardware_concurrency` : cpu가 지원하는 스레드 개수, `static` 멤버 함수
- `get_id` : 스레드 id 반환
- `native_handle` : OS의 스레드 핸들 반환
- `swap` : 스레드 객체 swap
- `joinable` : `join()` 가능한지 여부 조사

-
- join : 스레드 종료 대기

- detach : 스레드 떼어내기

hardware_concurrency 관련 예제를 살펴보자

```
1 void foo() {
2     std::cout << std::this_thread::get_id() << std::endl;
3 }
4
5 int main() {
6     int n = std::thread::hardware_concurrency();
7     std::cout << n << std::endl;
8
9     std::thread t(&foo);
10    std::this_thread::sleep_for(1s);
11
12    std::thread::id tid = t.get_id();
13    std::cout << "created thread id : " << tid << std::endl;
14    t.join();
15 }
```

native_handle 관련 예제를 살펴보자

```
1 #include <windows.h>           // 윈도우 시스템콜 함수 모음
2
3 void foo() {
4     auto tid = std::this_thread::get_id();
5     auto handle = GetCurrentThread();           // 핸들을 얻기 위한 시스템콜 함수
6     std::this_thread::sleep_for(1s);
7     std::cout << GetThreadPriority(handle) << std::endl; // 우선순위를 얻는 시스템콜 함수
8 }
9
10 int main() {
11     std::thread t(&foo);
12     std::thread::native_handle_type h = t.native_handle();
13
14     std::cout << "ID : " << t.get_id() << std::endl;
15     std::cout << "hanle : " << h << std::endl;
16
17     std::this_thread::sleep_for(100ms);
18     SetThreadPriority(h, THREAD_PRIORITY_TIME_CRITICAL); // 우선순위를 세팅하는 시스템콜 함수
19
20     t.join();
21 }
```

- `std::thread`에는 자체적으로 우선순위를 변경할 수 있는 메소드를 제공하지 않는다

- 운영체제(OS) 내에서 스레드 간 우선순위를 바꿀 수 있는데 `native_handle` 메소드를 호출하면 OS별로 핸들을 리턴해준다

- `native_handle_type` : 윈도우에서는 `handle`이며 리눅스에서는 `pthread handle`이다

join, joinable 예제를 살펴보자

```
1 int main() {
2     std::thread t;
3
4     if( t.joinable() ) {
5         t.join();
6     }
7 }
```

- `std::thread t`로 함수 인자를 넘기지 않았기 때문에 `join`이 불가능한 상황이다

- `joinable` 함수로 `join`이 가능한 상황인지 확인하고 `join` 함수를 실행한다

swap 관련 예제를 살펴보자

```

1 void foo() {}
2 void goo() {}

3
4 int main() {
5     std::thread t1(&foo);
6     std::thread t2(&goo);

7     t1.swap(t2);           // ok.

8
9     std::thread t3 = t1;      // error 복사 불가능
10    std::thread t4 = std::move(t1); // ok 이동은 가능

11
12    t1.join(); // error! 예외 발생
13    t2.join();
14    t3.join();
15
16 }

```

5.2.2 std::ref, std::reference_wrapper

```

1 void foo(int& a) { a = 200; }

2
3 template<typename T> void call_foo(T arg) {
4     foo(arg);
5 }

6
7 int main() {
8     int n = 0;

9
10    foo(n);           // good. n=200
11    call_foo(n);      // bad. n=0
12    call_foo( std::ref(n) ); // good. n=200
13 }

```

- call_foo는 인자를 T arg로 받고 있기 때문에 call_foo(n)은 복사본을 foo로 넘겨서 n의 값은 변하지 않는다
- std::ref(n)으로 넘기면 복사본이라고 하더라도 foo에 의해 n 값이 바뀐다. 왜 그럴까?
- std::ref : call-by-value을 사용하는 함수 템플릿에 객체를 참조로 보내고 싶을 때 사용하는 함수. <functional> 헤더가 필요하며 c++11부터 지원한다

우선 std::reference_wrapper의 원리를 먼저 살펴보자

```

1 template<typename T> struct reference_wrapper {
2     T* obj;
3     public:
4     reference_wrapper(T& t) : obj(&t) {}
5     operator T&() { return *obj; }
6 };
7
8 int main() {
9     int n = 0;
10    reference_wrapper<int> rw = n;
11
12    int& r = rw; // rw.operator int&() 호출
13    r = 100;
14
15    std::cout << n << std::endl; // 100
16 }

```

- std::reference_wrapper는 객체의 주소를 보관하고 있다가 필요할 때 참조(T&)로 암시적 변환이 가능한 클래스이다

모든 코드를 합쳐서 다시 작성해보자

```

1 template<typename T> struct reference_wrapper {

```

```

2     T* obj;
3     public:
4     reference_wrapper(T& t) : obj(&t) {}
5     operator T&() { return *obj; }
6 };
7
8 void foo(int& a) { a = 200; }
9
10 template<typename T> void call_foo(T arg) {
11     foo(arg);
12 }
13
14 int main() {
15     int n = 0;
16
17     reference_wrapper<int> rw = n;
18     call_foo(rw);
19
20     std::cout << n << std::endl; // 200
21 }
```

- `call_foo(rw)`가 호출되는 순간 `T arg`에서 `T`는 `std::reference_wrapper`가 된다
- `foo(arg)`가 호출될 때 `std::reference_wrapper`는 참조(`T&`)로 변환되므로 결론적으로 `n` 값이 참조로 넘어가는 것과 동일한 효과를 가진다
- 이를 보다 쉽게 사용하기 위해 `std::reference_wrapper`를 반환하는 `std::ref` 함수를 사용한다

```

1 namespace std {
2     template<typename T> reference_wrapper<T> ref(T& obj) {
3         return reference_wrapper<T>(obj);
4     }
5 }
6
7 ...
8
9 int main() {
10     int n = 0;
11
12     call_foo( std::ref(n) );
13
14     std::cout << n << std::endl; // 200
15 }
```

5.2.3 `std::promise`

여러 스레드에 작업을 시켰을 때 스레드1에서 스레드2로 결과를 전달하고 싶을 때가 있다. 이럴 때 사용하는 함수가 `std::promise`, `std::future`이다.

`std::promise`는 두 스레드 사이에서 값이나 예외를 공유할 수 있도록 해주는 함수이다. `<future>` 헤더가 필요하며 `std::promise`를 통해 전달된 데이터를 `std::future`를 통해 받을 수 있다

```

1 #include <future>
2
3 void add(std::promise<int>&& p, int a, int b) {
4     int s = a + b;
5     std::this_thread::sleep_for(1s);
6     p.set_value(s);           // future로 결과 전달
7 }
8
9 int main() {
10     std::promise<int> pm;
11     std::future<int> ft = pm.get_future();
12
13     std::thread t(add, std::move(pm), 10, 20);
14     //...
```

```

15
16     int ret = ft.get();           // promise로부터 결과 받음
17     std::cout << ret << std::endl;    // 30
18     t.join();
19 }
```

- 스레드 생성 시 `std::promise` 객체는 참조(&, &&) 형태로 전달되어야 한다
- `ft.get()`은 `p.set_value(s)`로부터 결과를 받을 때까지 대기한다
- `p.set_value_at_thread_exit(s)`를 사용하면 스레드가 종료될 때 값을 전달한다

`std::promise` 사용 시 예외가 발생하면 어떻게 처리해야 하는지 살펴보자.

```

1 void divide(std::promise<int>&& p, int a, int b) {
2     try{
3         if(b==0)
4             throw std::runtime_error("divide by zero");
5         p.set_value(a/b);
6     }
7     catch(...) {
8         p.set_exception( std::current_exception() );
9     }
10 }
11
12 int main() {
13     std::promise<int> pm;
14     std::future<int> ft = pm.get_future();
15
16     std::thread t(divide, std::move(pm), 10, 0);
17
18     try {
19         int ret = ft.get();
20     }
21     catch(std::exception& e) {
22         std::cout << e.what() << std::endl;
23     }
24
25     t.join();
26 }
```

- `p.set_exception` : 예외를 `future`에 던져줄 때 사용한다
- `p.set_exception_at_thread_exit` : 스레드가 종료될 때 예외를 전달한다

`std::promise`를 사용하여 두 스레드의 실행 타이밍을 맞추는 예제를 살펴보자

```

1 int main() {
2     std::vector<int> v1 = {1,2,3,4,4,5,6,7,8,9,10};
3     std::vector<int> v2(10);
4
5     // void형 promise 생성
6     std::promise<void> pm1;
7     std::promise<void> ft1 = pm1.get_future();
8
9     std::promise<int> pm2;
10    std::promise<int> ft2 = pm2.get_future();
11
12    std::thread t( [&] {
13        std::partial_sum(v1.begin(), v1.end(), v2.begin()); // #1 부분합 구하기
14
15        pm1.set_value(); // 1번 작업 종료됨을 알림
16
17        int s = std::accumulate(v2.begin(), v2.end(), 0); // #2 구간합 구하기
18
19        pm2.set_value(s); // 2번 작업의 값을 전달
20    });
21 }
```

```

22     ft1.get();    // 1번 작업이 종료될 때까지 대기
23     for(auto n : v2) {
24         std::cout << n << ", ";
25     }
26
27     int s = ft2.get(); // 2번 작업의 결과 전달
28     std::cout << "\n" << s << std::endl;
29
30     t.join();
31 }
```

5.2.4 std::future

`std::future`의 멤버함수를 살펴보자

<code>get</code>	결과를 꺼내기
<code>wait</code>	결과값이 준비될 때까지 대기
<code>wait_for</code>	주어진 시간만큼 대기
<code>wait_until</code>	주어진 시간까지 대기
<code>share</code>	<code>shared_future</code> 얻기
<code>valid</code>	유효성 확인

```

1 void add(std::promise<int>&& p, int a, int b) {
2     std::this_thread::sleep_for(3s);
3     p.set_value(a+b);
4 }
5
6 int main() {
7     std::promise<int> pm;
8     std::future<int> ft = pm.get_future();
9
10    std::thread t(add, std::move(pm), 10, 20);
11
12    // 2초간 대기한다
13    auto ret = ft.wait_for(2s);
14
15    if( ret == std::future_status::ready )
16        std::cout << "ready!" << std::endl;
17    else if( ret == std::future_status::time_out )
18        std::cout << "timeout!" << std::endl;
19    else
20        std::cout << "deferred!" << std::endl;
21 }
```

- `ft.wait_for`의 리턴 타입은 `std::future_status`이다
- `std::future_status::ready` : 결과값이 준비됨
- `std::future_status::timeout` : 시간 초과
- `std::future_status::deferred` : 연산을 수행할 함수가 아직 시작 안됨 (`async()` 함수에서 사용)

`std::shared_future` 개념에 대해 살펴보자

```

1 void add(std::promise<int>&& p, int a, int b) {
2     std::this_thread::sleep_for(1s);
3     p.set_value(a+b);
4 }
5
6 void consume(std::shared_future<int> sf) {
7     sf.get();
8     std::cout << "finish foo" << std::endl;
9 }
10
11 int main() {
12     std::promise<int> pm;
```

```

13     std::future<int> ft = pm.get_future();
14
15     std::future<int> ft2 = ft;           // error! 복사 불가능
16     std::shared_future<int> sft = ft.share();    // ok. 복사 가능
17     std::shared_future<int> sft = pm.get_future(); // ok. 처음부터 promise로부터 받아도 된다
18
19     std::thread t(add, std::move(pm), 10, 20);
20
21     std::thread t1(consume, sft);
22     std::thread t2(consume, sft);
23
24     t.join();
25     t1.join();
26     t2.join();
27 }
```

- `p.set_value(a+b)`로 전달되는 값을 여러 스레드에서 받고 싶은 경우
- `std::future` : 복사 불가능(noncopyable), 오직 이동만 가능(only movable)
- `std::shared_future` : 복사 가능(copyable)

`std::promise`, `std::future`를 사용할 때 주의점에 대해 살펴보자

```

1 void add(std::promise<int>&& p, int a, int b) {
2     std::this_thread::sleep_for(1s);
3     p.set_value(a+b);
4     p.set_value(a+b); // error! 한번만 사용 가능
5 }
6
7 int main() {
8     std::promise<int> pm;
9     std::future<int> ft = pm.get_future();
10    std::future<int> ft2 = pm.get_future(); // error! 한번만 사용 가능
11
12    std::thread t(add, std::move(pm), 10, 20);
13
14    int ret1 = ft.get();
15    int ret2 = ft.get(); // error! 한번만 사용 가능
16
17    if( ft.valid() ) { // valid로 future가 유효한지 확인
18        std::cout << "future is valid" << std::endl;
19    }
20
21    t.join();
22 }
```

- `std::promise`에서 `set_value()`는 한번만 사용할 수 있다. 그리고 `get_future()`도 한번만 사용할 수 있다
- `std::future`에서 `get()`도 한번만 사용할 수 있다

5.2.5 `std::packaged_task`

멀티스레드를 고려하지 않고 작성된 함수를 "비동기(스레드)로 실행"하려면 `std::packaged_task`를 사용해야 한다. `std::packaged_task`는 callable object(함수, 함수객체, 람다 등)을 비동기 호출(스레드로 호출) 할 수 있도록 래퍼를 만드는 도구이다.

```

1 // 스레드를 고려하지 않고 만든 함수
2 int add(int a, int b) {
3     std::cout << "add" << std::endl;
4     return a + b;
5 }
6
7 int main() {
8     std::packaged_task<int(int, int)> task(add);
9
10    std::future<int> ft = task.get_future();
```

```

11
12     std::thread t(std::move(task), 10, 20); // add(10,20)를 멀티스레드로 돌린 것과 동일함
13
14     int ret = ft.get();
15     std::cout << ret << std::endl;
16
17     t.join();
18 }
```

5.2.6 std::async

비동기 함수의 개념과 `std::async`의 개념에 대해 살펴보자.

- 동기 함수(synchronous function): 내부함수가 호출되면 반환될 때까지 다음 작업을 중단하는 함수
- 비동기 함수(asynchronous function): 내부함수가 호출되도 다음 작업을 계속 진행하는 함수
- 비동기 함수1 : "입출력(I/O) 작업을 수행"하는 비동기 함수 (`send(sock, data, ...)`). OS의 시스템 콜함수 (IOCP, EPOLL 등) 사용한다.
- 비동기 함수2 : "연산을 수행"하는 비동기 함수. 스레드를 만들어서 별도 작업을 한다(`std::thread`, `std::jthread`, `std::async`).

```

1 int add(int a, int b){
2     std::this_thread::sleep_for(2s);
3     return a+b;
4 }
5
6 int main() {
7     std::future<int> ft = std::async(add, 10, 20);
8
9     std::cout << "continue main" << std::endl;
10
11    int ret = ft.get(); // 2초간 대기 후 결과 반환
12    std::cout << "result: " << ret << std::endl;
13 }
```

- 함수를 비동기(스레드)로 실행하려면
- `std::thread`를 사용해서 스레드를 생성한다
- `std::jthread`를 사용한다(c++20)
- `std::async` 함수 템플릿을 사용한다. `std::async`는 주어진 함수를 비동기로 수행하는 함수 템플릿이다
- 기존에 작성된 함수를 간단하게 스레드로 실행할 수 있다
- 일반적인 구현은 스레드 풀을 사용하여 구현되어 있다
- `std::future`를 반환한다

```

1 int add(int a, int b){
2     std::cout << "add : " << std::this_thread::get_id() << std::endl;
3     std::this_thread::sleep_for(2s);
4     return a+b;
5 }
6
7 int main() {
8     std::future<int> ft = std::async( std::launch::async, add, 10, 20 );
9     std::future<int> ft = std::async( std::launch::deferred, add, 10, 20 );
10    std::future<int> ft = std::async( std::launch::async | std::launch::deferred, add, 10, 20 );
11    std::future<int> ft = std::async( add, 10, 20 );
12
13    std::cout << "continue main" << std::endl;
14
15    int ret = ft.get(); // 2초간 대기 후 결과 반환
16    std::cout << "result: " << ret << std::endl;
17 }
```

- `std::launch::async`: 새로운 스레드를 만들어 실행하는 옵션
- `std::launch::deferred`: 지연된 실행 옵션. 바로 실행하지 않고 `ft.get()`이 호출될 때 실행한다. 스레드는 동일 스레드를 사용한다
- `std::launch::async or std::launch::deferred` : 컴파일러에 따라 가능한 옵션 실행. PC용에서는 대부분 새로운 스레드 생성
- 옵션이 없다면 환경에 따라서 [새로운 스레드, 지연된 실행] 중 하나가 될 수 있다. PC용에서는 대부분 새로운 스레드 생성 (즉 위 or 코드와 동일하다)

`std::async`의 반환값에 대해 살펴보자

```

1 int add(int a, int b){
2     std::cout << "start add" << std::endl;
3     std::this_thread::sleep_for(2s);
4     std::cout << "end add" << std::endl;
5     return a+b;
6 }
7
8 int main() {
9     std::future<int> ft = std::async( std::launch::async, add, 10, 20 );
10
11    std::cout << "continue main" << std::endl;
12
13    // int ret = ft.get();           // get()이 없다면?
14
15    std::async( std::launch::async, add, 10, 20 ); // 반환값을 안받으면?
16 }
```

- `ft.get()`을 수행하지 않아도 `std::future`의 소멸자에서 `get()`을 호출하여 대기하고 있다
- 사용자가 명시적으로 `get()`을 호출하지 않아도 스레드 종료를 대기하게 된다
- `std::async`의 반환값을 받지 않는 경우 "임시 객체가 파괴될 때 소멸자에서 `get()`을 호출"한다
- 비동기(스레드)로 수행한 함수가 종료될 때까지 주 스레드가 대기하는 효과를 가진다(이 효과는 모든 `std::future`가 아닌 `std::async`의 반환값인 `std::future`에만 나타나는 것임에 유의한다)

`std::async`를 사용하는 간단한 예제를 살펴보자

```

1 static const int NUM = 100'000'000;
2
3 std::vector<int> v1, v2;
4
5 // 1억개 데이터 생성
6 void fill_vector(){
7     std::random_device seed;
8     std::mt19937 engine(seed());
9     std::uniform_distribution<int> dist(0, 100);
10
11    v1.reserve(NUM);
12    v2.reserve(NUM);
13
14    for(int i=0; i<NUM; i++){
15        v1.push_back(dist(engine));
16        v2.push_back(dist(engine));
17    }
18 }
19
20 // 1억개 데이터 내적을 단일스레드로
21 long long f1() {
22     return std::inner_product(v1.begin(), v1.end(), v2.begin(), 0LL);
23 }
24
25 // 1억개 데이터 내적을 4개의 스레드로
26 long long f2() {
27     auto future1 = std::async( [] { return std::inner_product(&v1[0], &v1[v1.size()/4], &v2[0], 0LL); });
28     auto future2 = std::async( [] { return std::inner_product(&v1[v1.size()/4], &v1[v1.size()/2], &v2[v1.size()/4], 0LL); });

```

```

29     auto future3 = std::async( [] { return std::inner_product(&v1[v1.size()/2], &v1[v1.size()*3/4],
30                                 &v2[v1.size()/2], 0LL); });
31     auto future4 = std::async( [] { return std::inner_product(&v1[v1.size()*3/4], &v1[v1.size()],
32                                 &v2[v1.size()*3/4], 0LL); });
33
34     return future1.get() + future2.get() + future3.get() + future4.get();
35 }
36
37 // 함수 실행 시간 측정
38 void measure_execution_time( std::string name, long long(*f)() ) {
39     std::chrono::system_clock::time_point start = std::chrono::system_clock::now();
40     long long result = f();
41     std::chrono::system_clock::time_point end = std::chrono::system_clock::now();
42
43     std::chrono::duration<double> time_span =
44         std::chrono::duration_cast<std::chrono::duration<double>>(end - start);
45
46     std::cout << name << " : " << result << ", " << time_span.count() << " seconds." << std::endl;
47 }
48
49 int main() {
50     fill_vector();
51     std::cout << "start inner product" << std::endl;
52
53     measure_execution_time("f1", f1);
54     measure_execution_time("f2", f2);
55 }
```

- g++ 기준 f1은 1.12초가 걸리고 f2는 0.1초가 걸림(12배 차이)

5.2.7 std::jthread

c++20부터 추가된 `std::jthread` 개념에 대해 살펴보자

```

1 void foo(int a, double d){
2     std::cout << " start foo " << std::endl;
3     std::this_thread::sleep_for(2s);
4     std::cout << " end foo " << std::endl;
5 }
6
7 int main() {
8     std::thread t(foo, 10, 3.4);
9     t.join();
10 }
```

- `std::thread` 사용 시 반드시 `join`, `detach`를 수행해야 한다
- 클래스에서 소멸자 호출 시 자동으로 `join`을 해줄 수는 없을까?

```

1 class mythread {
2     std::thread th;
3     public:
4     template<typename F, typename ... ARGS>
5     explicit mythread(F&& f, ARGS&& ... args)
6     : th(std::forward<F>(f), std::forward<ARGS>(args)...){}
7
8     ~mythread() {
9         if(th.joinable())
10             th.join();
11     }
12 };
13
14 int main() {
15     mythread t(foo, 10, 3.4);
16 }
```

c++20부터는 자동으로 소멸자에서 join()되는 스레드를 제공한다

```
1 int main() {
2     std::jthread t(foo, 10, 3.4);
3 }
```

- `std::jthread` : Cooperatively Interruptible + Joining Thread

- 1. Cooperatively Interruptible(협력적 중지 가능)

- 2. Joining Thread(소멸자에서 join 호출)

협력적으로 중지 가능하다는것이 어떤 뜻인지 살펴보자

```
1 void foo() {
2     for(int i=0; i<10; i++){
3         std::this_thread::sleep_for(500ms);
4         std::cout << "foo : " << i << std::endl;
5     }
6 }
7
8 void goo( std::stop_token token ) {
9     for(int i=0;i<10; i++) {
10         if(token.stop_requested()) {
11             std::cout << " stop request " << std::endl;
12             return;
13         }
14         std::this_thread::sleep_for(500ms);
15         std::cout << "goo : " << i << std::endl;
16     }
17 }
18
19 int main() {
20     std::jthread j1(foo);
21     std::jthread j2(goo);           // stop_token은 파라미터 생략 가능
22     std::this_thread::sleep_for(2s);
23
24     j1.request_stop();
25     j2.request_stop();
26 }
```

- `j2.request_stop`을 통해 함수 내부에 중지 요청을 할 수 있다

- 함수는 내부에서 `stop_token token`을 사용하여 중지 요청이 왔을 때 행동 양식을 작성할 수 있다

5.3 Synchronization

5.3.1 thread synchronization

스레드 동기화에 대해 살펴보자

```
1 void delay() { std::this_thread::sleep_for(20ms); }
2
3 void foo(std::string_view name) {
4     int x = 0;
5
6     for(int i=0; i<10; i++) {
7         x = 100; delay();
8         x = x+1; delay();
9         std::cout << name << " : " << x << std::endl; delay();
10    }
11 }
12
13 int main() {
14     std::thread t1(foo, "A");
15     std::thread t2(foo, "\tB");
16 }
```

```
17     t1.join();
18     t2.join();
19 }
```

- 지역변수 `int x`는 각 스레드 별 스택에 저장되므로 두 스레드 간 서로 충돌하지 않는다(지역변수는 스레드에 안전하다)

```
1 void delay() { std::this_thread::sleep_for(20ms); }
2
3 void foo(std::string_view name) {
4     static int x = 0;
5
6     for(int i=0; i<10; i++) {
7         x = 100; delay();
8         x = x+1; delay();
9         std::cout << name << " : " << x << std::endl; delay();
10    }
11 }
12
13 int main() {
14     std::thread t1(foo, "A");
15     std::thread t2(foo, "\tB");
16
17     t1.join();
18     t2.join();
19 }
```

- `static int x`는 data 메모리에 놓이기 때문에 모든 스레드가 공유한다(`static` 또는 전역변수는 스레드에 안전하지 않다)

```
1 std::mutex m;
2
3 void delay() { std::this_thread::sleep_for(20ms); }
4
5 void foo(std::string_view name) {
6     static int x = 0;
7
8     for(int i=0; i<10; i++) {
9         m.lock();
10        x = 100; delay();
11        x = x+1; delay();
12        std::cout << name << " : " << x << std::endl; delay();
13        m.unlock();
14    }
15 }
16
17 int main() {
18     std::thread t1(foo, "A");
19     std::thread t2(foo, "\tB");
20
21     t1.join();
22     t2.join();
23 }
```

- `std::mutex`의 `lock`, `unlock`을 사용하여 병렬 스레드를 직렬화할 수 있다

5.3.2 `std::mutex`, `std::timed_mutex`

c++ 표준이 제공하는 뮤텍스는 다음과 같다. 일반 버전과 `timed` 버전이 각각 3개씩 존재하여 총 6개가 존재한다.

- `std::mutex` (c++11)
- `std::recursive_mutex` (c++11)

- `std::shared_mutex` (c++17)
- `std::timed_mutex` (c++11)
- `std::recursive_timed_mutex` (c++11)
- `std::shared_timed_mutex` (c++14)

`std::mutex`의 멤버 함수는 다음과 같다

<code>lock</code>	locks the mutex, blocks if the mutex is not available
<code>try_lock</code>	tries to lock the mutex, return false if the mutex is not available
<code>unlock</code>	unlock mutex
<code>native_handle</code>	returns the underlying implementation-defined native handle object

멤버 타입은 다음과 같다

<code>native_handle_type</code>	<code>native_handle()</code> 멤버 함수의 반환 타입
---------------------------------	---

```

1   std::mutex m;
2   int share_data = 0;
3
4   void foo(){
5       if( m.try_lock() ) {      // lock 시도해보고 안되면 다른 행동 수행
6           share_data = 100;
7           std::cout << "using share_data" << std::endl;
8           m.unlock();
9       }
10      else {
11          std::cout << "fail to lock" << std::endl;
12      }
13  }
14
15  int main() {
16      std::thread t1(foo);
17      std::thread t2(foo);
18
19      t1.join();
20      t2.join();
21
22      std::mutex::native_handle_type h = m.native_handle();
23
24      std::mutex m2 = m; // error!
25 }
```

- `m.try_lock()`은 `lock`을 시도해보고 되지 않는다면 다른 행동을 수행하도록 해준다
- `native_handle`을 통해 OS의 고유 핸들을 얻을 수 있다
- `std::mutex`은 복사, 이동이 모두 불가능하다(non-copyable, non-movable)

`std::timed_mutex`에 대해 살펴보자

```

1   std::timed_mutex m;
2   int share_data = 0;
3
4   void foo(){
5       if( m.try_lock_for(2s) ) {      // 2초간 대기
6           share_data = 100;
7           std::cout << "using share_data" << std::endl;
8           std::this_thread::sleep_for(3s);      // 다른 스레드 진입 실패
9           m.unlock();
10      }
11      else {
12          std::cout << "fail to lock" << std::endl;
13      }
14 }
```

```
15
16     int main() {
17         std::thread t1(foo);
18         std::thread t2(foo);
19
20         t1.join();
21         t2.join();
22     }
```

- `std::timed_mutex`는 `try_lock_for`, `try_lock_until` 함수가 추가되었다
- 특정 시간동안 또는 시간까지 `lock`을 기다리는 행동을 할 수 있다

`std::recursive_mutex`에 대해 살펴보자

```
1  std::recursive_mutex m;
2  int share_data = 0;
3
4  void foo(){
5      m.lock();
6      m.lock(); // 2회 소유
7      share_data = 100;
8      std::cout << "using share_data" << std::endl;
9      m.unlock();
10     m.unlock();
11 }
12
13 int main() {
14     std::thread t1(foo);
15     std::thread t2(foo);
16
17     t1.join();
18     t2.join();
19 }
```

- `std::recursive_mutex` : 하나의 스레드가 "여러번 뮤텍스 소유" 가능. 소유한 횟수만큼 `unlock()`해야 한다

```
1  class Machine {
2      int shared_data = 0;
3      std::recursive_mutex m;
4  public:
5      void f1() {
6          m.lock();
7          shared_data = 100;
8          m.unlock();
9      }
10
11     void f2() {
12         m.lock();
13         shared_data = 200;
14         f1(); // 내부에 lock, unlock 있음
15         m.unlock();
16     }
17 };
18
19 int main() {
20     Machine m;
21
22     std::thread t1(&Machine::f1, &m);
23     std::thread t2(&Machine::f2, &m);
24
25     t1.join();
26     t2.join();
27 }
```

5.3.3 std::shared_mutex

c++17부터 등장한 `std::shared_mutex`에 대해 살펴보자

```
1 #include <shared_mutex>
2
3 std::shared_mutex m;
4 int share_data = 0;
5
6 void Writer() {
7     while(1){
8         m.lock();
9         share_data = share_data + 1;
10        std::cout << "Writer : " << share_data << std::endl;
11        std::this_thread::sleep_for(1s);
12        m.unlock();
13        std::this_thread::sleep_for(20ms);
14    }
15 }
16
17 void Reader(std::string_view name) {
18     while(1){
19         m.lock_shared();
20         std::cout << "Reader( " << name << ")" : " << share_data << std::endl;
21         std::this_thread::sleep_for(500ms);
22         m.unlock_shared();
23         std::this_thread::sleep_for(10ms);
24     }
25 }
26
27 int main() {
28     std::thread t1(Writer);
29     std::thread t2(Reader, "A");
30     std::thread t3(Reader, "B");
31     std::thread t4(Reader, "C"); // 여러개의 Reader 실행
32
33     t1.join();
34     t2.join();
35     t3.join();
36     t4.join();
37 }
```

- 쓰는 동안에는 읽을 수 없어야 한다
- 읽는 동안에는 쓸 수 없어야 한다
- 하나의 스레드가 읽는 동안에 다른 스레드도 읽을 수 있어야 한다
- `std::shared_mutex` : `lock_shared`, `unlock_shared` 메소드를 제공하여 서로 간섭하지 않는 스레드를 만들 수 있다
- `<shared_mutex>` 헤더가 필요하다

5.3.4 std::lock_guard

`std::lock_guard`란 도구에 대해 살펴보자

```
1 std::mutex m;
2
3 void goo() {
4     m.lock();
5     std::cout << "using shared data" << std::endl;
6     throw std::exception(); // 아래 unlock이 실행되지 않음
7     m.unlock();
8 }
9
10 void foo() {
11     try{
12         goo();
13     }
```

```

14     catch(...) {
15         std::cout << " catch exception" << std::endl;
16     }
17 }
18
19 int main() {
20     std::thread t1(foo);
21     std::thread t2(foo);
22
23     t1.join();
24     t2.join();
25 }
```

- `std::mutex` 사용 시 `lock`, `unlock`을 직접하는 경우 실수로 `unlock`을 하지 않을 수도 있다
- 코드 중간에 예외가 발생하면 `unlock`이 되지 않는다
- `std::lock_guard`: 생성자에서 `lock`, 소멸자에서 `unlock`을 수행하는 간단한 도구 (RAII)

```

1 void goo() {
2     std::lock_guard<std::mutex> lg(m);
3     std::cout << "using shared data" << std::endl;
4     throw std::exception(); // 예외가 발생해도 자동으로 unlock 호출
5 }
```

`std::lock_guard`의 구현 소스는 다음과 같다

```

1 template<class _Mutex>
2 class lock_guard {
3     public:
4         using mutex_type = _Mutex;
5
6         // 생성자 2개
7         explicit lock_guard(_Mutex& _Mtx) : _MyMutex(_Mtx) { _MyMutex.lock(); }
8         lock_guard(_Mutex& _Mtx, adopt_lock_t) : _MyMutex(_Mtx) {}
9
10        // 소멸자
11        ~lock_guard() noexcept { _MyMutex.unlock(); }
12
13        lock_guard(const lock_guard&) = delete;
14        lock_guard& operator=(const lock_guard&) = delete;
15
16        private:
17         _Mutex& _MyMutex;
18     };
19
20     struct adopt_lock_t {
21         explicit adopt_lock_t() = default;
22     };
23     constexpr adopt_lock_t adopt_lock {};
```

- 두번째 생성자는 이미 `lock`된 뮤텍스를 넘겨받을 때 사용한다

```

1 void foo() {
2     m.lock();
3     std::lock_guard<std::mutex> lg(m, std::adopt_lock);
4     ...
5 }
```

5.3.5 `std::unique_lock`

`std::unique_lock`이라는 도구에 대해 살펴보자. 기존의 `std::lock_guard`는 `lock`, `unlock`을 관리하는 최소한의 기능만 제공한다. 좀 더 다양한 방법으로 뮤텍스를 관리할 수 없을까? (`try_lock`, `lock_shared` 등 기능이 많은데) 뮤텍스의 `lock`, `unlock`을 관리하는 4개의 도구가 있다

<code>std::lock_guard</code>	한 개의 뮤텍스를 lock/unlock (c++11)
<code>std::unique_lock</code>	<code>std::lock_guard</code> 보다 다양한 기능을 제공 (c++11)
<code>std::scoped_lock</code>	여러개의 뮤텍스를 deadlock 없이 안전하게 lock/unlock (c++17)
<code>std::shared_lock</code>	<code>shared_mutex</code> 를 관리 (c++14)

```

1   std::mutex m1, m2, m3;
2   std::timed_mutex tm1, tm2, tm3;
3
4   int main() {
5     std::unique_lock<std::mutex> u1;
6     std::unique_lock<std::mutex> u2(m1);      // 생성자에서 m1.lock()
7
8     std::unique_lock<std::mutex> u3(m2, std::try_to_lock); // m2.try_lock();
9
10    if( u3.owns_lock() ) {
11      std::cout << "acquire lock" << std::endl;
12    }
13    else {
14      std::cout << "fail to lock" << std::endl;
15    }
16
17    m3.lock();
18    std::unique_lock<std::mutex> u4(m3, std::adopt_lock); // 이미 lock을 획득한 뮤텍스 관리
19
20    std::unique_lock<std::timed_mutex> u5(tm1, std::defer_lock); // 나중에 lock을 호출
21    auto ret = u5.try_lock_for(2s);
22
23    std::unique_lock<std::timed_mutex> u6(tm2, 2s);           //
24      tm2.try_lock_for()
25      사용
26  }

```

- `std::unique_lock`은 생성자가 다양하다
 - `std::unique_lock<std::mutex> u1;`
 - `std::unique_lock<std::mutex> u2(m);`
 - `std::unique_lock<std::mutex> u3(m, std::try_to_lock);`
 - `std::unique_lock<std::mutex> u4(m, std::adopt_lock);`
 - `std::unique_lock<std::mutex> u5(m, std::defer_lock);`
 - `std::unique_lock<std::mutex> u6(tm, 2s);`
 - `std::unique_lock<std::mutex> u7(tm, std::chrono::steady_clock::now() + 2s);`
- `std::unique_lock`의 다양한 멤버 함수에 대해 살펴보자

```

1   std::timed_mutex m;
2
3   int main() {
4     std::unique_lock<std::timed_mutex> u1;
5     std::unique_lock<std::timed_mutex> u2(m);
6
7     u1 = u2;                      // error! 복사 불가능
8     u1 = std::move(u2); // ok. 이동 가능
9
10    std::cout << u1.owns_lock() << std::endl;      // 1. lock을 획득했는지 확인 (0 or 1)
11
12    if( u1 ) {
13      std::cout << " acquire" << std::endl;
14    }
15    u1.unlock();
16
17    std::cout << u1.owns_lock() << std::endl;      // 0
18

```

```

19     if( u1.try_lock_for(3s) ) {
20         ...
21         u1.unlock();
22     }
23
24     u1.release();
25 }
```

5.3.6 std::lock, std::scoped_lock

```

1 class Account {
2     std::mutex m;
3     int money = 100;
4 };
5
6 void transfer(Account& acc1, Account& acc2, int cnt) {
7     acc1.m.lock();
8     std::this_thread::sleep_for(10ms);
9     acc2.m.lock();
10    acc1.money -= cnt;
11    acc2.money += cnt;
12    std::cout << "finish transfer" << std::endl;
13    acc2.m.unlock();
14    acc1.m.unlock();
15 }
16
17 int main() {
18     Account kim, lee;
19     std::thread t1(transfer, std::ref(kim), std::ref(lee), 5);
20     std::thread t2(transfer, std::ref(lee), std::ref(kim), 5);
21
22     t1.join();
23     t2.join();
24 }
```

- 두 개의 스레드가 두 개의 뮤텍스를 번갈아가며 lock하는 경우 서로 wait 상태에 들어가며 데드락 상태가 발생하게 된다

- `std::lock(acc1.m, acc2.m)`과 같이 STL 표준의 `std::lock` 함수를 사용하면 두 뮤텍스를 동시에 lock할 수 있다

```

1 void transfer(Account& acc1, Account& acc2, int cnt) {
2     std::lock(acc1.m, acc2.m);
3     acc1.money -= cnt;
4     acc2.money += cnt;
5     std::cout << "finish transfer" << std::endl;
6     acc2.m.unlock();
7     acc1.m.unlock();
8 }
```

- `std::lock` : 데드락 회피 기술을 사용해서 여러 개의 뮤텍스를 안전하게 lock하는 함수

```

1 void transfer(Account& acc1, Account& acc2, int cnt) {
2     std::lock(acc1.m, acc2.m);
3     std::lock_guard<std::mutex> lg1(acc1.m, std::adopt_lock);
4     std::lock_guard<std::mutex> lg2(acc2.m, std::adopt_lock);
5     acc1.money -= cnt;
6     acc2.money += cnt;
7     std::cout << "finish transfer" << std::endl;
8 }
```

- `std::lock_guard`와 `std::adopt_lock`을 사용하면 `unlock`을 명시적으로 작성해주지 않아도 된다

c++17부터 `std::scoped_lock`이 제공된다. `std::scoped_lock`는 여러개의 뮤텍스를 RAII 기술로 관리하는 함수를 말하며 내부적으로 `std::lock`을 사용한다

```
1 void transfer(Account& acc1, Account& acc2, int cnt) {
2     std::scoped_lock lg(acc1.m, acc2.m);
3     acc1.money -= cnt;
4     acc2.money += cnt;
5     std::cout << "finish transfer" << std::endl;
6 }
```

- `std::scoped_lock` : 두 개 이상의 뮤텍스를 데드락이 발생하지 않도록 보장하며 lock, unlock이 자동으로 이뤄지도록 하는 함수

5.3.7 `std::shared_lock`

```
1 std::shared_mutex m;
2 int share_data = 0;
3
4 void Writer() {
5     while(1){
6         {
7             std::lock_guard<std::shared_mutex> lg(m);
8             share_data = share_data + 1;
9             std::cout << "Writer : " << share_data << std::endl;
10            std::this_thread::sleep_for(1s);
11        }
12        std::this_thread::sleep_for(20ms);
13    }
14 }
15
16 void Reader(std::string_view name) {
17     while(1){
18         {
19             std::lock_guard<std::shared_mutex> lg(m); // error! shared mutex 특성이 사라져버림
20             std::cout << "Reader( " << name << " ) : " << share_data << std::endl;
21             std::this_thread::sleep_for(500ms);
22         }
23         std::this_thread::sleep_for(10ms);
24     }
25 }
26
27 int main() {
28     std::thread t1(Writer);
29     std::thread t2(Reader, "A");
30     std::thread t3(Reader, "B");
31     std::thread t4(Reader, "C"); // 여러개의 Reader 실행
32
33     t1.join();
34     t2.join();
35     t3.join();
36     t4.join();
37 }
```

- 이전 `std::shared_mutex` 예제를 `m.lock()`, `m.unlock()`을 제거하고 `std::lock_guard`를 사용하면 위 코드와 같다
- 하지만 Reader 부분에 `std::lock_guard`를 사용하면 `std::shared_mutex`의 특성이 사라져버리고 Reader들끼리도 직렬화된다

```
1 void Reader(std::string_view name) {
2     while(1){
3         {
4             std::shared_lock<std::shared_mutex> lg(m);
5             std::cout << "Reader( " << name << " ) : " << share_data << std::endl;
6             std::this_thread::sleep_for(500ms);
7 }
```

```

7     }
8     std::this_thread::sleep_for(10ms);
9 }
10 }
```

- `std::shared_lock`을 사용하면 Reader들끼리는 서로 lock되지 않고 빠르게 읽게 된다 (`std::shared_mutex` 기능과 동일)

5.3.8 `std::condition_variable`

c++ 표준의 `std::condition_variable`란 개념에 대해 살펴보자

```

1  std::mutex m;
2  int shared_data = 0;
3
4  void consumer() {
5      std::lock_guard<std::mutex> lg(m);
6      std::cout << "consume : " << shared_data << std::endl;
7 }
8
9  void producer() {
10     std::this_thread::sleep_for(10ms);
11     std::lock_guard<std::mutex> lg(m);
12     shared_data = 100;
13     std::cout << "produce : " << shared_data << std::endl;
14 }
15
16 int main() {
17     std::thread t1(producer);
18     std::thread t2(consumer);
19
20     t1.join(); t2.join();
21 }
```

- 공유 데이터(`shared_data`)를 생산자(`producer`)가 생산하고 소비자(`consumer`)가 소비하는 코드를 작성했다
- 가끔씩 생산자가 소비하기 전 소비자가 소비하는 경우가 발생한다
- 생산자가 생산 후 소비되어야 한다
- 데이터가 준비되었음을 알려야 한다 (`std::condition_variable`)

```

1 #include <condition_variable>
2
3 std::mutex m;
4 std::condition_variable cv;
5 int shared_data = 0;
6
7 void consumer() {
8     std::unique_lock<std::mutex> ul(m);
9     cv.wait( ul );           // 신호 대기
10
11     std::cout << "consume : " << shared_data << std::endl;
12 }
13
14 void producer() {
15     std::this_thread::sleep_for(10ms);
16     std::lock_guard<std::mutex> lg(m);
17     shared_data = 100;
18     std::cout << "produce : " << shared_data << std::endl;
19
20     cv.notify_one();        // 신호 전달
21 }
22
23 int main() {
24     std::thread t1(producer);
```

```
25     std::thread t2(consumer);
26
27     t1.join(); t2.join();
28 }
```

-
- `std::condition_variable` : <condition_variable> 헤더가 필요하다
 - `std::unique_lock`을 사용해야 한다

하지만 생산을 먼저하고 소비가 늦게 되는 경우 `cv.notify_one()`이 먼저 호출되고 `cv.wait(ul)`이 나중에 기다리므로 영원히 대기하게 된다.

```
1 #include <condition_variable>
2
3 std::mutex m;
4 std::condition_variable cv;
5 bool data_ready = false;
6 int shared_data = 0;
7
8 void consumer() {
9     std::this_thread::sleep_for(2s);
10    std::unique_lock<std::mutex> ul(m);
11    cv.wait( ul, [] { return data_ready; } );           // 조건자(predicate)를 사용하여 제어 가능
12
13    std::cout << "consume : " << shared_data << std::endl;
14 }
15
16 void producer() {
17     std::this_thread::sleep_for(10ms);
18     std::lock_guard<std::mutex> lg(m);
19     shared_data = 100;
20     data_ready = true;
21     std::cout << "produce : " << shared_data << std::endl;
22
23     cv.notify_one();
24 }
25
26 int main() {
27     std::thread t1(producer);
28     std::thread t2(consumer);
29
30     t1.join(); t2.join();
31 }
```

-
- `cv.wiat(ul, Predicate pred)`를 사용하여 소비자가 늦게 실행될 때에도 코드를 정상적으로 제어할 수 있다

데이터가 잘 소비(처리)됐는지 확인할 수 있는 `data_process` 변수를 추가해보자

```
1 #include <condition_variable>
2
3 std::mutex m;
4 std::condition_variable cv;
5 bool data_ready = false;
6 bool data_process = false;
7 int shared_data = 0;
8
9 void consumer() {
10     std::this_thread::sleep_for(2s);
11     std::unique_lock<std::mutex> ul(m);
12     cv.wait( ul, [] { return data_ready; } );
13
14     std::cout << "consume : " << shared_data << std::endl;
15
16     // 데이터 잘 소비했으면 다시 알림
17     data_process = true;
18     ul.unlock();
```

```

19     cv.notify_one();
20 }
21
22 void producer() {
23     std::this_thread::sleep_for(10ms);
24     std::lock_guard<std::mutex> lg(m);
25     shared_data = 100;
26     data_ready = true;
27     std::cout << "produce : " << shared_data << std::endl;
28
29     cv.notify_one();
30
31 // 데이터 잘 소비했는지 확인하는 코드 추가
32 std::unique_lock<std::mutex> ul(m);
33 cv.wait(ul, [] { return data_process; });
34 std::cout << "producer : data processed" << std::endl;
35 }
36
37 int main() {
38     std::thread t1(producer);
39     std::thread t2(consumer);
40
41     t1.join(); t2.join();
42 }
```

- producer --> consumer --> producer로 알림이 순차적으로 간다

만약 소비자가 한 명이 아니라 여러명이면 어떻게 해야 할까?

```

1 #include <condition_variable>
2
3 std::mutex m;
4 std::condition_variable cv;
5 bool data_ready = false;
6 int shared_data = 0;
7
8 void consumer() {
9     std::this_thread::sleep_for(2s);
10    std::unique_lock<std::mutex> ul(m);
11    cv.wait( ul, [] { return data_ready; } );
12
13    std::cout << "consume : " << shared_data << std::endl;
14 }
15
16 void producer() {
17     std::this_thread::sleep_for(10ms);
18     std::lock_guard<std::mutex> lg(m);
19     shared_data = 100;
20     data_ready = true;
21     std::cout << "produce : " << shared_data << std::endl;
22
23     cv.notify_all();
24 }
25
26 int main() {
27     std::thread t1(producer);
28     std::thread t2(consumer);
29     std::thread t3(consumer);
30     std::thread t4(consumer);
31
32     t1.join(); t2.join(); t3.join(); t4.join();
33 }
```

- cv.notify_one() : 대기중인 하나의 스레드를 깨운다
- cv.notify_all() : 대기중인 모든 스레드를 깨운다

5.3.9 std::semaphore

c++20부터 추가되는 `std::semaphore` 개념에 대해 살펴보자

```
1 std::mutex m;
2
3 void Download(std::string name) {
4     m.lock();
5     for(int i=0; i<100; i++){
6         std::cout << name;
7         std::this_thread::sleep_for(30ms);
8     }
9     m.unlock();
10 }
11
12 int main() {
13     std::thread t1(Download, "1");
14     std::thread t2(Download, "2");
15     std::thread t3(Download, "3");
16     std::thread t4(Download, "4");
17     std::thread t5(Download, "5");
18
19     t1.join(); t2.join(); t3.join(); t4.join(); t5.join();
20 }
```

- `std::semaphore`: 자원에 대한 한정적인 공유(n개의 스레드가 공유)를 하고 싶을 때 사용한다
- <semaphore> 헤더가 필요하다

```
1 std::counting_semaphore<3> sem(3)
2
3 void Download(std::string name) {
4     sem.acquire();
5     for(int i=0; i<100; i++){
6         std::cout << name;
7         std::this_thread::sleep_for(30ms);
8     }
9     sem.release();
10 }
11
12 int main() {
13     std::thread t1(Download, "1");
14     std::thread t2(Download, "2");
15     std::thread t3(Download, "3");
16     std::thread t4(Download, "4");
17     std::thread t5(Download, "5");
18
19     t1.join(); t2.join(); t3.join(); t4.join(); t5.join();
20 }
```

- `std::counting_semaphore<3> sem(3)` : 3개까지 스레드를 공유한다(최대값 3, 카운터 3)
- `std::counting_semaphore<MAX_COUNT> sem(counter init value)`

```
1 // sem.acquire()
2 if(sem.count) -sem.counter;
3 else wait sem.count > 0;
4
5 // sem.release(update=1)
6 sem.counter += update
7 "update < 0" or
8 "sem.counter + update > MAX" 라면
9 std::system_error 예외 발생
```

mutex	자원의 독점 lock 획득한 스레드만 unlock 가능
semaphore	자원의 한정적인 공유 모든 스레드가 counter를 증가할 수 있다

```

1 ...
2
3 int main() {
4     std::thread t1(Download, "1");
5     std::thread t2(Download, "2");
6     std::thread t3(Download, "3");
7     std::thread t4(Download, "4");
8     std::thread t5(Download, "5");
9
10    std::this_thread::sleep_for(2s);
11    std::cout << " main " << std::endl;
12    sem.release();           // 모든 스레드에서 release를 할 수 있다
13
14    t1.join(); t2.join(); t3.join(); t4.join(); t5.join();
15 }
```

```

1 std::counting_semaphore<3> sem(0) // 0으로 해놓으면 다른 곳에서 release할 때까지 대기한다는 의미
2
3 std::counting_semaphore<1> sem(1) // 최대1, 카운트 1로 해놓으면 기본 뮤텍스와 비슷하게 하나의 스레드만
4     자원을 독점할 수 있다
5
6 std::binary_semaphore sem(1);           // alias 버전
7
8 - using binary_semaphore = counting_semaphore<1>;
9
10 std::semaphore의 멤버함수는 다음과 같다
```

acquire	decrements the internal counter or blocks until it can
try_acquire	tries to decrements the internal counter without blocking
try_acquire_for	tries to decrement the internal counter; blocking for up to a duration time
try_acquire_until	tries to decrement the interal counter; blocking until a point in time
release	increments the internal counter and unblocks acquires

5.3.10 std::latch, std::barrier

c++20에서 추가된 `std::latch`에 대해 살펴보자

```

1 void foo(std::string name) {
2     std::cout << "start work : " << name << std::endl;
3     std::cout << "finish work : " << name << std::endl;
4     std::cout << "go home : " << name << std::endl;
5 }
6
7 int main() {
8     std::jthread t1(foo, "kim"), t2(foo, "lee"), t3(foo, "park");
9 }
```

- 세 명의 일꾼이 작업이 끝나면 각자 go home하지 말고 주 스레드에 보고 후 같이 go home하도록 해보자
- `std::latch` : 카운트 기반의 스레드 동기화 도구
- 내부적으로 `std::atomic`을 사용하는 간단한 클래스

```

1 #include <latch>
2
3 std::latch complete{3};
4 std::latch gohome{1};
```

```

6 void foo(std::string name) {
7     std::cout << "start work : " << name << std::endl;
8     std::cout << "finish work : " << name << std::endl;
9
10    complete.count_down(); // --count
11    gohome.wait();
12
13    std::cout << "go home : " << name << std::endl;
14 }
15
16 int main() {
17     std::jthread t1(foo, "kim"), t2(foo, "lee"), t3(foo, "park");
18
19     complete.wait(); // 카운트가 0일때까지 대기
20
21     std::cout << "receive signal" << std::endl;
22
23     gohome.count_down();
24 }
```

세 개의 스레드가 모두 도착한 후에 다음 작업을 수행하려면 어떻게 해야할까?

```

1 std::latch sync_point{3};
2
3 void foo(std::string name) {
4     std::cout << "start work : " << name << std::endl;
5     std::cout << "finish work : " << name << std::endl;
6
7     sync_point.arrive_and_wait(); // 카운트가 0이 될때까지 대기하겠다
8
9     std::cout << "go home : " << name << std::endl;
10 }
11
12 int main() {
13     std::jthread t1(foo, "kim"), t2(foo, "lee"), t3(foo, "park");
14 }
```

- `sync_point.arrive_and_wait()` : 도착했으니 카운트를 1개 뺀 후 0이 될때까지 대기한다

```

1 #include <barrier>
2
3 std::barrier sync_point{3};
4
5 void foo(std::string name) {
6     std::cout << "start work : " << name << std::endl;
7     std::cout << "finish work : " << name << std::endl;
8
9     sync_point.arrive_and_wait();
10    std::cout << "have dinner: " << name << std::endl;
11
12    sync_point.arrive_and_wait(); // barrier일때만 다시 사용 가능!
13    std::cout << "go home : " << name << std::endl;
14 }
15
16 int main() {
17     std::jthread t1(foo, "kim"), t2(foo, "lee"), t3(foo, "park");
18 }
```

- 모두 일이 끝나고 `have dinner`하고 `go home`하려면 두 개의 `std::latch`가 필요하다
- `std::latch`의 카운트는 한 번밖에 사용하지 못한다
- `std::barrier`를 사용하면 카운트를 다시 사용할 수 있다

`std::barrier`는 콜백함수를 등록할수도 있다

```

1 #include <barrier>
2
3 // 클백함수
4 void oncomplete() {
5     std::cout << "oncomplete" << std::endl;
6 }
7
8 std::barrier sync_point{3, oncomplete};
9
10 void foo(std::string name) {
11     std::cout << "start work : " << name << std::endl;
12     std::cout << "finish work : " << name << std::endl;
13
14     sync_point.arrive_and_wait();           // 마지막으로 도착한 스레드가 oncomplete 호출
15     std::cout << "have dinner: " << name << std::endl;
16
17     sync_point.arrive_and_wait();           // 마지막으로 도착한 스레드가 oncomplete 호출
18     std::cout << "go home : " << name << std::endl;
19 }
20
21 int main() {
22     std::jthread t1(foo, "kim"), t2(foo, "lee"), t3(foo, "park");
23 }
```

5.3.11 thread_local

스레드 로컬 스토리지(thread local storage)라는 개념에 대해 살펴보자

```

1 int next3times() {
2     static int n = 0;
3     n = n + 3;
4     return n;
5 }
6
7 void foo(std::string_view name) {
8     std::cout << name << " : " << next3times() << std::endl;
9     std::cout << name << " : " << next3times() << std::endl;
10    std::cout << name << " : " << next3times() << std::endl;
11 }
12
13 int main() {
14     foo("A");
15 }
```

- 3의 배수를 차례대로 반환하는 함수 만들기
- `static int n`을 사용하여 함수 호출이 끝나도 지역변수가 파괴되지 않도록 한다
- `static` 방법은 싱글스레드일 때는 안전하지만 멀티스레드 환경에서는 안전하지 않다

```

1 int next3times() {
2     static int n = 0;
3     n = n + 3;
4     return n;
5 }
6
7 void foo(std::string_view name) {
8     std::cout << name << " : " << next3times() << std::endl;
9     std::cout << name << " : " << next3times() << std::endl;
10    std::cout << name << " : " << next3times() << std::endl;
11 }
12
13 int main() {
14     std::thread t1(foo, "A");
15     std::thread t2(foo, "\tB");    // 멀티스레드 사용
16 }
```

```
17     t1.join();
18     t2.join();
19 }
```

-
- 1. 지역변수 : 스레드 당 한 개 함수 호출이 끝날 때마다 파괴된다
 - 2. **static** 지역변수 : 모든 스레드가 공유하지만 멀티스레드일 때 안전하지 않다
 - 3. **TLS(Thread Local Storage)** : 스레드 당 한 개씩 할당된 **static** 지역변수라고 생각하면 된다. 멀티스레드에도 서로 독립적으로 존재하기 때문에 안전하다

TLS를 사용하려면 다음 키워드를 사용해야 한다

- Linux(gcc) : `_thread static int x`
- Windows(cl) : `_declspec(thread)`
- C++ 표준 : `thread_local`

```
1 int next3times() {
2     thread_local int n = 0;
3     n = n + 3;
4     return n;
5 }
6
7 void foo(std::string_view name) {
8     std::cout << name << " : " << next3times() << std::endl;
9     std::cout << name << " : " << next3times() << std::endl;
10    std::cout << name << " : " << next3times() << std::endl;
11 }
12
13 int main() {
14     std::thread t1(foo, "A");
15     std::thread t2(foo, "\tB");
16
17     t1.join();
18     t2.join();
19 }
```

-
- **thread_local** : 변수를 TLS에 저장해달라는 키워드
 - **static**을 표기하지 않아도 암시적으로 **static**로 취급된다

5.3.12 `std::call_once`, `std::once_flag`

```
1 void init(int a, double d) {
2     std::cout << "init" << std::endl;
3 }
4
5 void foo() {
6     std::cout << "start foo" << std::endl;
7     init(10, 3.4); // 한 번만 호출하고 싶다
8     std::cout << "finish foo" << std::endl;
9 }
10
11 int main() {
12     std::thread t1(foo);
13     std::thread t2(foo);
14     std::thread t3(foo);
15
16     t1.join(); t2.join(); t3.join();
17 }
```

-
- `foo`는 세 개의 스레드에서 세 번 호출되지만 초기화(`init`) 함수는 한 번만 호출하고 싶은 경우 어떻게 해야할까?
 - `std::call_once()`를 사용하면 된다. <`mutex`> 헤더가 필요하다

```

1 #include <mutex>
2
3 std::once_flag init_flag;
4
5 void init(int a, double d) {
6     std::cout << "init" << std::endl;
7 }
8
9 void foo() {
10    std::cout << "start foo" << std::endl;
11    std::call_once(init_flag, init, 10, 3.4);
12    std::cout << "finish foo" << std::endl;
13 }
14
15 int main() {
16     std::thread t1(foo);
17     std::thread t2(foo);
18     std::thread t3(foo);
19
20     t1.join(); t2.join(); t3.join();
21 }
```

- `bool` 플래그를 사용하여 유저가 직접 코딩할 수도 있으나 `std::once_flag`는 **복사와 이동을 모두 삭제(=delete)** 해놔서 간단하게 사용할 수 있다
- `std::call_once`에서 초기화 함수가 오래 걸리는 경우 다른 스레드가 다음 줄로 넘어가지 않고 `std::call_once`가 마무리될 때까지 기다린다 (바라는 동작)

`std::call_once`를 사용하는 싱글톤 코드 예제를 살펴보자

```

1 class Singleton {
2     private:
3     Singleton() = default;
4     static Singleton* sinstance;
5     public:
6     Singleton(const Singleton&) = delete;
7     Singleton& operator=(const Singleton&) = delete;
8
9     static Singleton* getInstance() {
10         if(sinstance == nullptr) {
11             sinstance = new Singleton;
12         }
13         return sinstance;
14     }
15 };
16 Singleton* Singleton::sinstance = nullptr;
17
18 int main() {
19     std::cout << Singleton::getInstance() << std::endl;
20     std::cout << Singleton::getInstance() << std::endl;
21 }
```

- 싱글스레드 환경에서 위 싱글톤 코드는 안전하지만 멀티스레드 환경에서는 `sinstance == nullptr` 구문에서 여러 스레드가 동시에 인스턴스를 생성할 수 있으므로 안전하지 않다

```

1 class Singleton {
2     private:
3     Singleton() = default;
4     static Singleton* sinstance;
5     static std::once_flag create_flag;
6     public:
7     Singleton(const Singleton&) = delete;
8     Singleton& operator=(const Singleton&) = delete;
9
10    static Singleton* getInstance() {
```

```

11     std::call_once(create_flag, initSingleton); // 멀티스레드에서도 한번만 호출됨
12     return sinstance;
13 }
14 static void initSingleton() { // 인스턴스 생성 함수 따로 작성
15     sinstance = new Singleton;
16 }
17 };
18 Singleton* Singleton::sinstance = nullptr;
19 std::once_flag Singleton::create_flag;

```

Meyer's singleton 패턴도 살펴보자

```

1 class Singleton {
2     private:
3     Singleton(){
4         std::cout << "start ctor" << std::endl;
5         std::this_thread::sleep_for(3s);
6         std::cout << "finish ctor" << std::endl;
7     }
8     public:
9     Singleton(const Singleton&) = delete;
10    Singleton& operator=(const Singleton&) = delete;
11
12    static Singleton& getInstance() {
13        std::cout << "start getInstance" << std::endl;
14        static Singleton instance;           // Meyer's singleton pattern
15        std::cout << "finish getInstance" << std::endl;
16
17        return instance;
18    }
19 };
20
21 void foo(){
22     Singleton& s = Singleton::getInstance();
23     std::cout << &s << std::endl;
24 }
25
26 int main() {
27     std::thread t1(foo);
28     std::thread t2(foo);
29     std::thread t3(foo);
30
31     t1.join(); t2.join(); t3.join();
32 }

```

- Meyer의 싱글톤 패턴은 `static` 변수를 사용하여 객체를 생성한다
- `c++11부터 static 변수는 멀티스레드에 안전`하기 때문에 `static Singleton instance` 부분에서 객체의 생성자가 호출 종료될 때까지 다른 스레드도 자동으로 대기한다
- 따라서 `std::once_flag`, `std::call_once`가 필요하지 않다

5.4 atomic

5.4.1 `std::atomic`

원자 연산이라고 불리는 `std::atomic` 연산에 대해 살펴보자

```

1 long x = 0;
2
3 void foo() {
4     for(int i=0; i<100000; i++) {
5         ++x;
6     }
7 }
8
9 int main() {

```

```
10     std::thread t1(foo);
11     std::thread t2(foo);
12     std::thread t3(foo);
13
14     t1.join(); t2.join(); t3.join();
15
16     std::cout << x << std::endl;
17 }
```

- x를 10만번씩 세 개의 스레드가 작업하기 때문에 이상적인 결과값으로는 30만이 나와야 한다

++x의 어셈블리 코드를 보자

```
1 mov    eax, x
2 add    eax, 1
3 mov    x, eax
```

- x는 모든 스레드가 공유하는 변수

- 첫번째 스레드가 add eax, 1을 지날 때 두번째 스레드가 mov eax, x를 지나가면 유효하지 않은 값을 끼내게 되는 것이므로 값이 엉키게 된다

첫번째 해결책은 OS가 제공하기도는 동기화 도구를 사용하는 방법이다(`std::mutex`)

```
1 std::mutex m;
2 long x = 0;
3
4 void foo() {
5     for(int i=0; i<100000; i++) {
6         m.lock();
7         ++x;
8         m.unlock();
9     }
10 }
```

- 정상적으로 작동하지만 이러한 간단한 연산을 하기 위해 `std::mutex`까지 사용하기에는 무겁다 (속도가 느린다)

두번째 해결책은 cpu가 제공하는 스레드(멀티코어 환경)에 안전한 명령어(OPCODE)를 사용하는 것이다. 인텔의 경우 lock prefix를 사용한다

```
1 long x = 0;
2
3 void foo() {
4     for(int i=0; i<100000; i++) {
5         __asm
6         {
7             lock inc x
8         }
9     }
10 }
```

- g++은 사용할 수 없고 윈도우 컴파일러(cl)에서만 작동한다

- 이러한 명령어들을 더 이상 조절 수 없는 연산이라는 의미로 원자 연산(atomic operation)이라고 부른다

윈도우에서는 별도의 원자 연산 wrapper가 존재한다

```
1 #include <windows.h>
2
3 long x = 0;
4
5 void foo() {
6     for(int i=0; i<100000; i++) {
7         InterlockedIncrement(x);
8     }
9 }
```

-
- 윈도우 전용이기 때문에 리눅스에서는 사용할 수 없다
 - `std::atomic`은 운영체제 단이 아닌 언어 단에서 원자 연산을 사용할 수 있도록 해준다

```

1 #include <atomic>
2
3 std::atomic<long> x = 0;
4
5 void foo() {
6     for(int i=0; i<100000; i++) {
7         ++x;           // x.operator++() 연산자 재정의
8     }
9 }
```

`std::atomic`은 다음 연산을 제공한다

연산자 재정의 함수	<code>operator++ operator-- operator+= operator-= operator&= operator == operator^=</code>
멤버함수	<code>fetch_add fetch_sub fetch_and fetch_or fetch_xor</code>

```

1 #include <atomic>
2
3 std::atomic<long> x = 0;
4
5 void foo() {
6     for(int i=0; i<100000; i++) {
7         x.fetch_add(1);    // fetch_add도 동일한 효과를 갖는다
8     }
9 }
```

- `fetch_add`는 두번째 파라미터를 추가할 수 있다
- `x.fetch_add(1, std::memory_order_relaxed)`
- 두번째 파라미터를 명시하지 않으면 `std::memory_order_seq_cst`가 사용된다

`memory_order` 종류는 다음과 같다. 자세한 설명은 별도의 섹션에서 한다

```

1 typedef enum memory_order {
2     memory_order_relaxed,
3     memory_order_consume,
4     memory_order_acquire,
5     memory_order_release,
6     memory_order_acq_rel,
7     memory_order_seq_cst,
8 } memory_order;
```

`std::atomic`에 대한 또 다른 예제를 살펴보자

```

1 struct Point {int x, y;};
2 struct Point3D {int x, y, z;};
3
4 std::atomic<int> at1;
5 std::atomic<Point> at2;
6 std::atomic<Point3D> at3;
7
8 int main() {
9     ++at1;
10
11     std::cout << at1.is_lock_free() << std::endl; // 1. ok
12     std::cout << at2.is_lock_free() << std::endl; // 1. ok
13     std::cout << at3.is_lock_free() << std::endl; // 0. no
```

 }

- lock-free : OS의 동기화 도구를 사용하지 않고 CPU 레벨의 명령어를 사용해서 동기화하는 것 (더욱 안전)
- int는 lock-free이지만 사용자가 만든 구조체도 lock-free일까?
- at2는 32비트 int가 2개있으므로 64비트 구조체가 되어 CPU에서 멀티코어에 안전하게 load, store하는 것이 가능하다(=lock-free)
- at3는 96비트이기 때문에 lock-free가 지원되지 않는다

또 다른 예제를 살펴보자

```

1 struct Point {
2     int x,y;
3     Point() = default;
4
5     Point(const Point&) {} // 복사생성자가 있으면 load()할 때 에러가 발생한다
6 };
7
8 std::atomic<Point> pt;
9
10 int main() {
11     Point ret = pt.load();
12 }
```

- 복사 생성자가 없으면 코드가 잘 컴파일되지만 복사 생성자가 있으므로 함수에 lock이 걸려서 atomic 연산이 되지 않고 컴파일 에러가 발생한다

- std::atomic<T> : T의 복사계열의 함수와 move 계열의 함수가 모두 trivial해야 한다

```

1 template<class _Ty>
2 struct atomic{
3     static_assert(
4         is_trivially_copyable_v<_Ty> &&
5         is_copy_constructible<_Ty> &&
6         is_move_constructible<_Ty> &&
7         is_copyAssignable_v<_Ty> &&
8         is_moveAssignable_v<_Ty> &&,
9         "atomic<T> required T to be trivially copyable, copy constructible, move constructible, copy
10         assignable, "
11         "and move assignable.");
12 }
```

- trivial : 사용자가 임의로 만든 것이 아닌 디폴트 생성자

```

1 struct Point {
2     int x,y;
3     Point() = default;
4
5     Point(const Point&) = default; // 복사생성자가 trivial하므로 잘 컴파일된다
6 };
7
8 std::atomic<Point> pt;
9
10 int main() {
11     Point ret = pt.load();
12 }
```

5.4.2 reordering

```

1 int a=0;
2 int b=0;
3
```

```

4 // thread A
5 void foo() {
6     a = b+1;
7     b = 1;           // b가 1이 되기 전 a에 값을 넣으므로 a는 1이어야 한다
8 }
9
10 // thread B
11 void goo(){
12     if(b==1) {
13         // a==1을 보장할 수 있을까?
14     }
15 }
```

어셈블리 소스 코드를 만드는 방법은 다음과 같다

```

1 g++ reorder1.cpp -S -masm=intel
2 cl.exe reorder1.cpp /FAs /c
```

- 어셈블리 코드를 확인해보면 $a=1$ 이 되는 것을 확인할 수 있다
- 하지만 최적화를 수행하면 어떻게 될까? $g++ -O2$, $cl.exe /O2$ 옵션을 줘보자
- 최적화 어셈블리 코드를 보면 $b=1$ 입력이 $a=b+1$ 보다 먼저 들어가게 되므로 값이 정상적으로 나오지 않을 수 있다

CPU에는 Cache 공간과 Register 공간이 있다.

```

1 CPU = [ Cache, Register(eax, ebx, ...) ]
2 Memory = [ a, b ]
```

- $a=b+1$ 에서 b 를 Cache에 올리고 연산 후 b 를 내리고 a 를 올리는 과정이 필요하다. 그리고 $b=1$ 가 호출되면 다시 b 를 Cache에 올리는 과정이 비효율적이기 때문에 최적화 과정에서는 이 순서를 지키지 않는다
- b 가 이미 Cache에 있으므로 a 에 결과를 넣기 전에 $b=1$ 을 먼저 실행하면 성능 향상을 볼 수 있지 않을까?
- 이러한 최적화를 **reordering**이라고 한다. 성능 향상을 위해 코드의 실행 순서를 변경하는 것을 말한다. 컴파일 시간, 실행 시간에 모두 발생한다.

이러한 코드 최적화 버그를 해결하기 위해서는 코드 사이에 펜스(fence)를 쳐서 둘을 확실하게 갈라주는 것이 필요하다.

```

1 void foo(){
2     a = b + 1;
3     // ----- fence -----
4     __asm { mfence }
5     b = 1;
6 }
```

- **mfence**는 인텔 명령어이고 CPU마다 다르기 때문에 C++ 표준이 필요하다

```

1 void foo(){
2     a = b + 1;
3     // ----- fence -----
4     std::atomic_thread_fence( std::memory_order_release );
5     b = 1;
6 }
```

5.4.3 std::memory_order

```

1 std::atomic<int> x=0;
2 std::atomic<int> y=0;
3
4 void foo() {
5     int n1 = y.load();
6     x.store(n1);
```

```

7   }
8
9   void goo() {
10    int n2 = x.load();
11    y.store(100);
12 }
13
14 int main() {
15    std::thread t1(foo);
16    std::thread t2(goo);
17    t1.join(); t2.join();
18 }
```

- 멀티스레드 코드를 보면 고려해야 할 사항은 다음과 같다
- 주어진 표현식이 스레드에 안전하게 실행되는가?(atomic operator이 가능한가?)
- reordering이 발생하지 않는가?
- 원자 연산에서 값을 가져올 때는 load()를 사용하고 저장할 때는 store()를 사용한다
- load(), store()는 memory_order 옵션을 별도로 지정해줄 수 있다

```

1 void foo() {
2    int n1 = y.load( std::memory_order_relaxed );
3    x.store(n1, std::memory_order_relaxed);
4 }
5
6 void goo() {
7    int n2 = x.load( std::memory_order_relaxed );
8    y.store(100, std::memory_order_relaxed);
9 }
```

- `std::memory_order_relaxed` : 가장 오버헤드가 적다
- atomic operation만 보장해주며 실행순서는 변경될 수 있다

또 다른 예제를 살펴보자

```

1 std::atomic<int> data1 = 0;
2 std::atomic<int> data2 = 0;
3 std::atomic<int> flag = 0;
4
5 void foo() {
6    data1.store(100, std::memory_order_relaxed);
7    data2.store(200, std::memory_order_relaxed);
8    flag.store(1, std::memory_order_release);
9 }
10
11 void goo() {
12    if( flag.load( std::memory_order_acquire ) > 0 ) {
13        assert(data1.load(std::memory_order_relaxed) == 100);
14        assert(data2.load(std::memory_order_relaxed) == 200);
15    }
16 }
17
18 int main() {
19    std::thread t1(foo);
20    std::thread t2(goo);
21    t1.join(); t2.join();
22 }
```

- goo에서 flag의 값을 보고 0보다 크면 data1, data2의 값이 제대로 저장됐는지 확인한다
- `std::memory_order_relaxed`는 순서를 보장하지 않기 때문에 이를 사용하면 안전하지 않다
- release-acquire 모델 : `release 이전의 코드는 acquire 이후에 읽을 수 있다는 것을 보장`해야 한다

또 다른 예제를 보자

```
1 std::atomic<int> data1 = 0;
```

```

2     std::atomic<int> data2 = 0;
3
4     int main() {
5         data1.store( 100, std::memory_order_seq_cst );
6         data2.store( 200, std::memory_order_seq_cst );
7         data2.store( 300 );
8     }

```

- `std::memory_order_seq_cst` : 원자 연산과 순서를 모두 보장한다
- 파라미터를 생략하면 디폴트로 사용되는 옵션이다
- 여러 옵션 중 가장 오버헤드가 크다

5.4.4 `std::atomic_flag`

```

1     std::mutex m;
2
3     void work(){
4         m.lock();
5         std::cout << " start. using shared resource" << std::endl;
6         std::cout << " end. using shared resource" << std::endl;
7         m.unlock();
8     }
9
10    int main() {
11        std::jthread t1(work), t2(work);
12    }

```

- `mutex`를 사용하여 두 스레드가 공유 자원에 접근할 때 충돌하지 않도록 작성한 코드
- `mutex` 말고 다른 방법이 있을까?

```

1     bool use_flag = false;
2
3     void work(){
4         while( use_flag );
5         use_flag = true;      // 사용 중
6         std::cout << " start. using shared resource" << std::endl;
7         std::cout << " end. using shared resource" << std::endl;
8         use_flag = false;    // 사용 끝
9     }
10
11    int main() {
12        std::jthread t1(work), t2(work);
13    }

```

- **busy waiting** 기술 : 자원 접근 권한을 얻기 위해 대기(sleeping)하는 것이 아니라 **루프를 돌면서 확인**하는 방법
- CPU 자원을 낭비하므로 일반적인 상황에서는 좋지 않은 방식
- 자원의 접근 권한을 얻기 위해 대기 시간이 짧은 경우에 주로 사용한다 (예제 코드 같은 경우)
- 예제 코드는 `use_flag`가 서로 동기화되어 있지 않으므로 멀티스레드 코드에서 안전하지 않다. 결국에는 `std::atomic<bool>`을 사용해야 한다

`std::atomic<bool>`을 대신하여 C++에서는 `std::atomic_flag`를 사용할 수 있다. 이는 `std::atomic<bool>`과 유사하지만 **lock-free를 보장**한다(OS가 아닌 CPU 명령어를 사용한다)

```

1     std::atomic_flag = ATOMIC_FLAG_INIT;           // until c++20
2     std::atomic_flag flag;                         // c++20. false로 초기화
3
4     void work(){
5         while( flag.test_and_set() );
6
7         std::cout << " start. using shared resource" << std::endl;
8         std::cout << " end. using shared resource" << std::endl;
9

```

```

10     flag.clear(); // flag = false;
11 }
12
13 int main() {
14     std::jthread t1(work), t2(work);
15 }
```

- `std::atomic_flag` : lock-free를 보장한다
- 최소의 멤버함수만 제공한다 (`store`, `load`가 없다)
- `flag.test_and_set()`: 하나의 스레드를 제외하고 나머지 스레드는 `true`가 되어 무한 루프로 대기를 한다(spin lock이라고도 함)

`std::atomic_flag`를 사용한 간단한 클래스 예제를 살펴보자

```

1 class spinlock{
2     std::atomic_flag = flag;
3     public:
4     void lock() { while(flag.test_and_set()); }
5     void unlock() {flag.clear(); }
6 };
7 spinlock spin;
8
9 void work() {
10    spin.lock();
11    std::cout << " start. using shared resource" << std::endl;
12    std::cout << " end. using shared resource" << std::endl;
13    spin.unlock();
14 }
15
16 int main() {
17     std::jthread t1(work), t2(work);
18 }
```

- busy wait를 활용한 클래스 `spinlock`을 만들어 마치 `mutex`처럼 동작하도록 작성하였다

5.4.5 `std::atomic_ref`

c++20부터 지원하는 `std::atomic_ref` 개념에 대해 살펴보자

```

1 struct Machine{
2     int data{ 0 };
3     int count{ 0 };
4 };
5 Machine m;
6
7 void foo() {
8     for(int i=0; i<1000000; i++){
9         ++(m.count);
10    }
11 }
12
13 int main() {
14 {
15     std::jthread t1(foo), t2(foo), t3(foo);
16 }
17 std::cout << m.count << std::endl;
18 }
```

- c++20부터 참조(reference)처럼 동작하는 atomic 클래스가 추가되었다
- 위 코드는 `m.count`가 동기화되어 있지 않기 때문에 300만이라는 값이 나오지 않는다
- `m.count`를 동기화시키기 위해 `std::atmoic<int>` `count{0}`과 같이 사용해도 되지만 `std::atomic_ref`를 사용하면 더욱 간단하게 문제를 해결할 수 있다

```

1 void foo() {
2     std::atomic_ref<int> cnt{m.count};
3     for(int i=0; i<1000000; i++){
4         ++cnt;
5     }
6 }
```

5.5 STL

5.5.1 STL parallel algorithm

c++17부터 지원되는 STL의 병렬 알고리즘에 대해 살펴보자

```

1 #include <execution>
2
3 void foo(int n){
4     std::cout << n << " : " << std::this_thread::get_id() << std::endl;
5 }
6
7 int main() {
8     std::vector<int> v{1,2,3,4,5,6,7,8,9,10};
9
10    std::for_each(v.begin(), v.end(), foo);           // 단일스레드 호출이므로 같은 id가 출력됨
11
12    std::for_each(std::execution::par, v.begin(), v.end(), foo); // 멀티스레드로 호출하라
13 }
```

- STL 병렬 알고리즘은 c++17부터 지원하며 69개 알고리즘을 지원한다

<code>std::execution::seq</code>	싱글 스레드
<code>std::execution::par</code>	알고리즘을 병렬로 실행
<code>std::execution::par_unseq</code>	알고리즘을 병렬로 실행(vectorized, SIMD)
<code>std::execution::unseq</code>	싱글 스레드(vectorized, SIMD), c++20부터 지원

병렬 알고리즘을 사용할 때 주의사항에 대해 살펴보자

```

1 int main() {
2     int vector<int> v(100, 0);
3
4     for(int i=1; i<=100; i++) v.push_back(i);
5
6     int sum = 0;
7
8     std::for_each(std::execution::par, v.begin(), v.end() [&](int n){
9         sum += n;
10        std::this_thread::sleep_for(1ms);
11    });
12 }
```

- `sum += n`에서 예상값은 5050이 나와야 한다
- 하지만 `int sum`은 동기화되어 있지 않기 때문에 총돌이 나서 다른 값이 나온다

```

1 int main() {
2     int vector<int> v(100, 0);
3
4     for(int i=1; i<=100; i++) v.push_back(i);
5
6     std::atomic<int> sum = 0;
7
8     std::for_each(std::execution::par, v.begin(), v.end() [&](int n){
9         sum.fetch_add(n, std::memory_order_relaxed );
10        std::this_thread::sleep_for(1ms);
11    });
12 }
```

```
11     });
12 }
```

- STL 병렬 알고리즘을 사용할 때 공용 변수가 있으면(`sum`) 충돌이 일어나기 쉬우므로 원자 연산을 사용하는 것이 빠르고 안전하다

5.5.2 atomic smart pointer

c++20부터 추가된 atomic smart pointer에 대해 살펴보자

```
1 void foo(){
2     std::shared_ptr<int> ptr = std::make_shared<int>(5);
3
4     // 값에 의한 캡처(복사본이 만들어진다)
5     std::thread t1( [ptr]() mutable{
6         ptr = std::make_shared<int>(1);
7     });
8
9     std::thread t2( [ptr]() mutable{
10        ptr = std::make_shared<int>(2);
11    });
12    t1.join(), t2.join();
13 }
14
15 int main() {
16     foo();
17 }
```

- `t1, t2` 스레드가 생성될 때 `ptr`의 복사본이 생성되므로 `ptr`은 서로 다른 세 개의 스레드(`main, t1, t2`)에 각각 존재하게 된다

- `shared_ptr`은 서로 다른 스레드에서도 `ref`를 공유할까?
- `shared_ptr`은 설계 단계부터 멀티스레드 환경에서도 자원을 공유하도록 설계되었다

```
1 void goo(){
2     std::shared_ptr<int> ptr = std::make_shared<int>(5);
3
4     // 참조에 의한 캡처
5     std::thread t1( [&]() mutable{
6         ptr = std::make_shared<int>(1);
7     });
8
9     std::thread t2( [&]() mutable{
10        ptr = std::make_shared<int>(2);
11    });
12    t1.join(), t2.join();
13 }
```

- 참조에 의한 캡처를 하면 `ptr`이 세 개의 스레드 모두 동일한 포인터가 된다
- 이 때 `t1, t2`에서 다른 곳을 가리키면 `ptr`은 어떻게 될까?
- `ptr`은 멀티스레드 환경에서 안전하지 않다 (한 개의 포인터를 세 개의 스레드에서 공유하므로)

```
1 void goo(){
2     std::atomic<std::shared_ptr<int>> ptr = std::make_shared<int>(5);
3
4     // 참조에 의한 캡처
5     std::thread t1( [&]() mutable{
6         ptr = std::make_shared<int>(1);
7     });
8
9     std::thread t2( [&]() mutable{
10        ptr = std::make_shared<int>(2);
11    });
12 }
```

```
12     t1.join(), t2.join();
13 }
```

- `std::atomic<std::shared_ptr<int>>` ptr을 사용하면 t1, t2에서 사용하는 ptr이 스레드에 안전해진다
- 정리하면 다음과 같다
- `std::shared_ptr` 스마트 포인터의 참조 계수 증가, 감소는 멀티스레드 환경에서 안전하다
- 하지만 `std::shared_ptr` 자체는 스레드에 안전하지 않다
- c++20부터 `std::atomic`에 스마트 포인터를 넣어서 사용이 가능하다

6 Template programming

6.1 Template basic

6.1.1 c++ template intro

템플릿의 기본 개념에 대해 살펴보자

```
1 int square(int a){  
2     return a*a;  
3 }  
4  
5 double square(double a){  
6     return a*a;  
7 }  
8  
9 int main() {  
10     square(3);  
11     square(3.3);  
12 }
```

- 함수 오버로딩(function overloading): 인자의 형태가 다르면 동일 이름의 함수를 여러 개 만들 수 있다 (`square(int)`, `square(double)`)

- 장점: 함수 이름이 동일하므로 사용자 입장에서는 하나의 함수처럼 사용하게 된다

- 단점: `square` 함수를 만들 때 인자 타입과 반환 타입만 다르고 구현이 동일한 함수를 여러 개 만들어야 한다

- C++ 언어의 해결책 : 구현이 동일한 함수가 여러 개 필요하면 함수를 만들지 말고 함수를 생성하는 틀(템플릿)을 만들자

```
1 template<typename T>  
2 int square(T a) {  
3     return a*a;  
4 }  
5  
6 int main() {  
7     square<int>(3);  
8     square<double>(3.3);  
9 }
```

- 함수 앞에 `template<typename T>`를 추가한다

- 컴파일 시 호출된 타입을 사용해서 컴파일러가 실제 함수를 생성한다 (**템플릿 인스턴스화, template instantiation라고 부른다**)

함수 템플릿을 사용하는 방법

템플릿 인자를 명시적으로 전달	<code>square<int>(3);</code> 사용자가 전달한 타입으로 함수를 생성
템플릿 인자를 생략	<code>square(3);</code> 컴파일러가 함수 인자를 보고 타입을 추론(type deduction)

6.1.2 View template instantiation

템플릿 인스턴스화 결과를 확인하는 방법은 다음과 같다

(1)	컴파일 결과로 생성된 어셈블리 코드를 확인 godbolt.org 사이트 (compiler explorer)
(2)	cppinsights.io 사이트
(3)	템플릿 인스턴스화의 결과로 생성된 함수의 이름을 출력

첫번째 방법은 [godbolt.org \(compiler explorer\)](https://www.godbolt.org)를 사용하는 방법이다. 다양한 언어의 컴파일 결과를 어셈블리 코드로 확인할 수 있다

템플릿 자체는 컴파일시간에 컴파일러가 함수를 생성하기 위해서만 사용된다. 함수 템플릿을 만들고 사용하지 않으면 인스턴스화.instantiation이 되지 않는다. 실제 함수는 생성되지 않는다.
템플릿 함수에 대해 다양한 타입이 호출되면 실행 파일의 크기가 커지는 현상이 발생하는데 이를 **코드 폭발(code bloat)**라고 한다(템플릿 함수가 너무 많이 인스턴스화.instantiation이 되어 코드 메모리가 증가하는 현상)

두번째 방법은 cppinsight.io 사이트에서 확인하는 방법이다. 이 사이트는 c++ 코드의 다양한 내부 원리를 보여주는 사이트이다(template, range-for, virtual function 등의 원리를 알고 싶을 때 사용)

세번째 방법은 인스턴스화 된 함수 이름을 출력하는 방법이다.

```
1 #include <source_location>
2
3 template<typename T>
4 T square(T a) {
5     std::cout << __FUNCTION__ << std::endl;           // #1 cl 컴파일러
6     std::cout << __PRETTY_FUNCTION__ << std::endl; // #2 g++, clang 컴파일러
7
8     std::source_location s = std::source_location::current();
9     std::cout << s.function_name() << std::endl; // #3
10
11    return a*a;
12 }
13
14 int main() {
15     square<int>(3);
16     square<double>(3.3);
17     square(3);
18 }
```

- `__FUNCTION__` : c++ 표준 매크로. 시그니처가 포함되어 있지 않은 함수 이름만 있다
- `__FUNCSIG__`: 비표준 매크로. cl 컴파일러 전용이며 시그니처가 포함되어 있다
- `__PRETTY_FUNCTION__`: 비표준 매크로. g++, clang 컴파일러 전용이며 시그니처가 포함되어 있다
- `std::source_location`: c++20부터 지원하는 클래스. 파일 이름, 라인 번호, 함수 이름 등을 구할 수 있다

6.1.3 template misc

몇가지 템플릿 기본 문법을 알아보자

```
1 template<typename T>
2 T square1(T a) {
3     return a*a;
4 }
5
6 template<class T>          // typename = class 동일
7 T square2(T a) {
8     return a*a;
9 }
10
11 auto square3(auto a) {    // 템플릿 키워드가 없음에도 템플릿으로 분류됨(c++20부터 사용)
12     return a*a;
13 }
14
15 int main() {
16     square1<int>(3);
17     square2<int>(3);
18     square3<int>(3);
19
20     // 인자 생략 가능
21     square1(3);
22     square2(3);
23     square3(3);
24 }
```

함수와 함수 템플릿의 차이점을 살펴보자

```

1 template<typename T>
2 T square(T a){
3     return a*a;
4 }
5
6 int main() {
7     printf("%p\n", &square);           // error! 템플릿은 그 자체로 함수가 아니므로 에러가 발생한다
8     printf("%p\n", &square<int>);    // ok. 타입을 특정해주는 순간 템플릿이 함수가 된다
9 }
```

템플릿 코드를 분할하여 저장한 경우에 대해 살펴보자

```

1 // square.h
2 template<typename T>
3 T square(T a);
4
5 // square.cpp
6 template<typename T>
7 T square(T a){
8     return a*a;
9 }
10
11 // using_square.cpp
12 #include "square.h"
13 int main(){
14     square<int>(3);      // error! 이 순간 square<int>를 생성하려면 템플릿의 구현을 컴파일러가 알아야 한다
15 }
```

- 템플릿은 구현부를 헤더 파일에 작성해야 한다 (또는 명시적 인스턴스화를 해야한다)

템플릿에 대한 다양한 관점들에 대해 살펴보자

1. 함수(클래스)를 생성하는 틀(template)
2. 함수(클래스)들의 군(family)
3. 함수(클래스)의 일반화 버전(generic)

템플릿의 종류는 4가지가 있다

1. function template
2. class template
3. variable template
4. using template(template alias)

6.1.4 explicit template instantiation

템플릿의 명시적 인스턴스화(explicit instantiation)에 대해 살펴보자

```

1 template<typename T>
2 void fn(T a) {}
3
4 template void fn<int>(int); // 명시적 인스턴스화
5 template void fn<>(double); // 타입 생략 가능
6 tamplate void fn(char); // 꺥쇠도 생략 가능
```

- 함수를 아직 main에서 사용하지 않았음에도 인스턴스화되어 코드 메모리에 올라간다

클래스 템플릿도 가능하다

```

1 template<class t>
2 class Type{
3     void mf1() {}
4     void mf2() {}
5 };
6
7 template class Type<int>; // int로 전체 인스턴스화
8 template void Type<double>::mf1(); // mf1만 double로 인스턴스화

```

- 템플릿 인스턴스화: 템플릿으로부터 실제 함수, 클래스를 생성하는 과정
- 암시적 인스턴스화와 명시적 인스턴스화가 존재한다

implicit instantiation	<code>fn<int>(3); fn(3);</code>
explicit instantiation	<code>template void fn<int>(int); template void fn<>(int); template void fn(int); template class Type<int>; template void Type<double>::mf1();</code>

명시적 인스턴스화를 하는 간단한 예제에 대해 살펴보자

```

1 // square.h
2 template<typename T>
3 T square(T a);
4
5
6 // square.cpp
7 template<typename T>
8 T square(T a){
9     return a*a;
10 }
11
12 template int square(int);
13 template double square(double);
14
15 // using_square.cpp
16 #include "square.h"
17 int main(){
18     square(3);
19     square(3.3);
20 }

```

- 템플릿의 일반적인 코딩 관례는 함수 템플릿의 구현부를 헤더 파일에 포함하는 것이다
- 다른 방법은 cpp 파일 안에 명시적 인스턴스화를 하면 컴파일이 가능하다 (코드 보안성 향상)

6.1.5 function & function template

함수와 함수 템플릿에 대해 살펴보자

```

1 // 함수 템플릿
2 template<typename T>
3 T square(T a){
4     std::cout << "T" << std::endl;
5     return a*a;
6 }
7
8 // 함수
9 int square(int a){
10    std::cout << "int" << std::endl;
11    return a*a;

```

```

12 }
13
14 int main() {
15     square(3);           // int. 타입 생략 시 함수템플릿이므로 int 버전 함수가 호출된다
16     square(3.4);        // T
17
18     square<int>(3);    // T
19     square<>(3);       // T
20 }
```

- 함수 `square`와 함수 템플릿 `square<int>`가 이름 충돌이 나는지 궁금할 수 있다
- 하지만 둘은 함수 이름이 다르다

- `square(int)` : _Z6squarei: (g++), ?square@@YAHHC@Z (cl)
- `square<int>(int)` : _Z6squareIiET_S0_ : (g++), ??@square@H@CYAZHH@Z (cl)

c++20에 추가된 concept을 사용한 오버로딩 기법에 대해 알아보자

```

1 #include <concept>
2
3 template<typename T> requires std::integral<T>
4 T square(T a){
5     std::cout << "integral" << std::endl;
6     return a*a;
7 }
8
9 int main() {
10     square(3);          // ok
11     square(3.4);        // error!
12 }
```

- 인자 타입(`int`, `double`)별로 구현을 다르게 하고 싶은 경우가 있다
- `requires std::integral<T>`: T가 정수(`int`) 계열일 때만 해당 템플릿을 사용하겠다

```

1 // 정수가 아닌 계열일 때 사용하는 템플릿
2 template<typename T> requires (!std::integral<T>)
3 T square(T a){
4     std::cout << "not integral" << std::endl;
5     return a*a;
6 }
```

6.1.6 template return type

함수 템플릿과 리턴 타입에 대해 살펴보자

```

1 template<typename T>
2 T add(const T& a, const T& b){
3     return a+b;
4 }
5
6 int main() {
7     std::cout << add(3, 4) << std::endl;      // int
8     std::cout << add(3.1, 4.3) << std::endl; // double
9
10    std::cout << add(3, 4.3) << std::endl; // ?
11 }
```

- `add`가 서로 다른 타입을 받기 위해서는 어떻게 해야 할까?

```

1 template<typename T1, typename T2>
2 ? add(const T1& a, const T2& b){
```

```

3     return a+b;
4 }
5
6 int main() {
7     std::cout << add(3, 4.3) << std::endl; // int, double
8 }
```

- 템플릿의 `typename`을 서로 다르게 하면 다른 타입을 받을 수 있다
- 그렇다면 반환 타입은 어떻게 결정해야 할까?

```

1 // #1 사용자가 직접 템플릿 인자로 전달
2 template<typename R, typename T1, typename T2>
3 R add(const T1& a, const T2& b){
4     return a+b;
5 }
6
7 // #2 auto, decltype 사용
8 template<typename T1, typename T2>
9 auto add(const T1& a, const T2& b) -> decltype(a+b) { // c++11
10    return a+b;
11 }
12
13 auto add(const T1& a, const T2& b) { // c++14 (후위 반환 타입 생략 가능)
14    return a+b;
15 }
16
17 // #3 type_traits 기술 사용 (std::common_type)
18 #include <type_traits>
19 typename std::common_type<T1, T2>::type addadd(const T1& a, const T2& b) {
20    return a+b;
21 }
```

- `decltype(a+b)`: a,b로부터 리턴 타입을 추론하는 키워드
- 리턴 타입에 a,b가 들어가므로 선언 뒤쪽에 들어가야한다(후위 반환 타입, c++11)
- c++14에는 후위 반환 타입 생략이 가능하다
- `typename std::common_type<T1, T2>::type` : T1,T2의 공통 타입을 사용하겠다

c++20에서는 더 간단하게 `auto`만 사용해서 작성 가능하다

```

1 auto add(const auto& a, const auto& b){
2     return a+b;
3 }
4
5 int main(){
6     std::cout << add(3, 4.3) << std::endl;
7 }
```

6.1.7 class template

이전 섹션에서 함수 템플릿에 대해 알아봤다면 이번 섹션부터는 클래스 템플릿에 대해 살펴보자

```

1 template<typename T>
2 class Vector{
3     T* ptr;
4     std::size_t size;
5 public:
6     Vector(std::size_t sz) : size(sz) {
7         ptr = new T[sz];
8     }
9     ~Vector() { delete[] ptr; }
10
11     T& operator[](std::size_t idx) { return ptr[idx]; }
12 }
```

```

13     Vector(const Vector&) = delete;
14 };
15
16 void fn(const Vector<int>& v) {}
17
18 int main() {
19     Vector v<int>(10);
20     Vector v(10);           // c++17부터는 타입 생략이 가능하다
21
22     v[0] = 10;
23     fn(v);
24 }
```

- Vector : 템플릿의 이름
- Vector<type> : 클래스(타입)의 이름
- 복사 생성자를 지우는 클래스 내 코드 Vector(const Vector&) = delete는 타입을 생략해도 된다

클래스 템플릿의 멤버 함수를 외부에서 구현하는 코드를 살펴보자

```

1 template<typename T>
2 class Vector{
3     T* ptr;
4     std::size_t size;
5 public:
6     Vector(std::size_t sz);
7
8     ~Vector();
9
10    T& operator[](std::size_t idx) { return ptr[idx]; }
11
12    Vector(const Vector&) = delete;
13 };
14
15 // 생성자 외부 구현
16 template<typename T>
17 Vector<T>::Vector(std::size_t sz) : size(sz) {
18     ptr = new T[sz];
19 }
20
21 // 소멸자 외부 구현
22 template<typename T>
23 Vector<T>::~Vector() { delete[] ptr; }
```

- 외부 구현 코드에도 template<typename T>를 붙여줘야 한다

또 다른 예제를 살펴보자

```

1 template<typename T1, typename T2>
2 struct Object{
3     T1 first;
4     T2 second;
5     static int cnt;
6
7     Object(const T1& a, const T2& b);
8 };
9
10 template<typename T1, typename T2>
11 Object<T1, T2>::Object(const T1& a, const T2& b)
12     :first(a), second(b) {}
13
14 // static 변수의 외부 구현
15 template<typename T1, typename T2>
16 int Object<T1, T2>::cnt = 0;
17
18 int main() {
```

```
19     Object<int, double> obj1(1, 3.4);
20 }
```

6.1.8 member template

클래스의 멤버함수 템플릿에 대해 살펴보자

```
1 template<typename T>
2 class Object{
3     public:
4         void mf1(int n) {}
5         void mf2(T n) {}
6
7         template<typename U>
8         void mf3(U n) {}
9     };
10
11 int main() {
12     Object<int> obj1;
13     Object<double> obj2;
14
15     obj1.mf1(3);
16     obj2.mf1(3);
17
18     obj1.mf2(3);      // T = int
19     obj2.mf2(3.4);   // T = double
20
21     obj1.mf3(3);
22     obj2.mf3(3.4);
23 }
```

- `mf1`, `mf2`는 클래스 템플릿의 멤버함수이다. **함수 자체는 템플릿이 아님**
- `mf3`는 멤버 함수 템플릿이다. **함수 자체가 템플릿이다**

멤버 함수 템플릿을 외부에서 구현하려면 복잡한 모양이 된다

```
1 template<typename T> template<typename U>
2 void Object<T>::mf3(U n)
```

멤버 함수 템플릿을 사용하는 예제를 살펴보자

```
1 template<typename T>
2 class Point{
3     T x, y;
4     public:
5     Point(const T& a, const T& b) : x(a), y(b) {}
6 };
7
8 int main() {
9     Point<int> p1(1,2);    // ok
10    Point<int> p2 = p1;   // ok
11
12    Point<double> p3 = p1; // ??
13 }
```

- `Point<double>`과 `Point<int>`는 다른 클래스이기 때문에 복사 생성자가 동작하지 않는다

```
1 template<typename T>
2 class Point{
3     T x, y;
4     public:
5     Point(const T& a, const T& b) : x(a), y(b) {}
6
7     // 서로 다른 타입의 Point들끼리 복사 생성자를 호출하는 코드
```

```

8     template<typename U>
9     Point(const Point<U>& p) : x(p.x), y(p.y) {}
10
11    template<typename> friend class Point; // 다른 클래스의 private에 접근하기 위해 friend 사용
12 };

```

클래스 템플릿에서 사용할 수 있는 복사 생성자들은 다음과 같다

Point(const Point& p)	복사 생성자
Point(const Point<T>& p)	자신과 동일한 타입만 받을 수 있다
Point(const Point<int>& p)	생성자 Point<int> 타입만 받을 수 있다
template<typename U> Point(const Point<U>& p)	generic (복사) 생성자 임의 타입의 Point를 받을 수 있다

마지막 기법의 이름을 **Coercion by member template**이라고 부른다. T가 U로 복사(대입)이 가능하다면 Point<T>도 Point<U>로 복사(대입)이 가능해야 한다

다음으로 main 문을 살펴보자

```

1 #include <type_traits>
2
3 template<typename T>
4 class Point{
5     T x, y;
6     public:
7     Point(const T& a, const T& b) : x(a), y(b) {}
8
9     template<typename U> requires std::is_convertible_v<U,T>           // 변환 가능한 경우만 사용
10    Point(const Point<U>& p) : x(p.x), y(p.y) {}
11
12    template<typename> friend class Point;
13 };
14
15 int main(){
16     Point<std::string> p1("1", "2");
17
18     Point<int> p2 = p1;      // error! p1의 string 객체가 int로 복사될 수 없다
19 }

```

- string은 int로 변환되지 않으므로 아예 복사 생성자를 호출하지 않는 것이 낫다
- requires 키워드를 사용하여 변환 가능한 타입만 복사 생성자를 호출하도록 한다

STL의 스마트 포인터와 관련된 예제를 살펴보자

```

1 class Animal { int age; }
2 class Dog : public Animal {};
3
4 int main() {
5     std::shared_ptr<Dog> p1(new Dog);
6
7     std::shared_ptr<Animal> p2 = p1; // 스마트 포인터가 다른 타입을 가리켜야 할 때
8 }

```

- Dog를 Animal 포인터로 가리킬 수 있으나 스마트 포인터 shared_ptr는 클래스이므로 서로 다른 타입을 가리키는 형태가 된다
- 그럼에도 불구하고 p2는 p1을 가리킬 수 있어야 한다
- 따라서 shared_ptr에는 앞서 설명한 generic (복사) 생성자가 반드시 구현되어 있어야 한다

6.1.9 lazy instantiation

지연된 인스턴스화라는 개념에 대해 살펴보자

```

1 class Object{
2     int value;
3 public:
4     void mf(){
5         *value = 10;      // error! int이므로 dereference 불가
6     }
7 };
8
9 int main() {
10     Object obj;
11 }
```

```

1 template<typename T>
2 class Object{
3     T value;
4 public:
5     void mf(){
6         *value = 10;      // ok
7             - 128) * 64 + (' - 128) * 64 + (' - 128) * 64 + (' - 128) * 64 + (' - 128)
8             - 128) * 64 + (' - 128) * 64 + (' - 128) * 64 + (' - 128) * 64 + (' - 128)
9             - 128) * 64 + (' - 128) * 64 + (' - 128) * 64 + (' - 128)
10    }
11 };
12
13 int main() {
14     Object<int> obj;
15 }
```

- **지연된 인스턴스화(lazy instantiation)**: 클래스 템플릿의 멤버함수는 사용된 것만 인스턴스화 된다
- main문에서 mf() 함수는 호출된 적이 없기 때문에 실제 코드 메모리 상에 올라오지 않았다(따라서 컴파일이 정상적으로 됨)

```

1 int main() {
2     Object<int> obj;
3     obj.mf();        // 함수를 호출했으므로 인스턴스화 된다(에러 발생)
4 }
```

또 다른 예제를 살펴보자

```

1 template<typename T, typename C> class queue{
2     C c;
3 public:
4     void push(const T& a) { c.push_back(a); }
5     void pop() { c.pop_front(); }
6 };
7
8 int main() {
9     queue<int, std::vector<int>> q;
10    q.push(10);
11 }
```

- `std::vector`는 `pop_front()` 함수가 존재하지 않는다
- 하지만 main에서 `pop`을 호출하지 않았기 때문에 인스턴스화 되지 않아서 컴파일 에러가 발생하지 않는다

```

1 int main() {
2     queue<int, std::vector<int>> q;
3     q.push(10);
4     q.pop();        // 인스턴스화되어 에러 발생
5 }
```

- C++ 표준의 `std::queue` 또한 동일한 문제를 가지고 있다
- `std::queue`의 두 번째 인자로 `std::vector`를 사용하지 말라고 권고하는데 `push()`만 사용하는 경우 지연된 인스턴

스화로 인해 컴파일 에러가 발생하지 않아서 사용해도 되는 것으로 오해할 수 있다
- 하지만 pop()을 호출하는 즉시 컴파일 에러가 발생하므로 사용하지 않는 것이 좋다

또 다른 예제를 살펴보자

```
1 struct static_member{
2     static_member() { std::cout << "static_member()" << std::endl; }
3     ~static_member() { std::cout << "~static_member()" << std::endl; }
4 };
5
6 struct instance_member{
7     instance_member() { std::cout << "instance_member()" << std::endl; }
8     ~instance_member() { std::cout << "~instance_member()" << std::endl; }
9 };
10
11 struct Object{
12     instance_member m1;
13     static static_member m2;
14 };
15 static_member Object::m2;
16
17 int main() {
18     std::cout << "-----" << std::endl;
19     Obj obj;
20     std::cout << "-----" << std::endl;
21 }
```

- 클래스의 `static` 멤버 데이터는 **객체를 생성하지 않아도 메모리에 놓인다**(`static`멤버가 유저 정의 타입이면 생성자 호출)
- 따라서 main문에서 첫번째 -----가 나오기 전에 이미 `static_member`의 생성자가 불린다
- 실행 결과
- `static_member`
- -----
- `instance_member`
- -----
- `~instance_member`
- `~static_member`

만약 `Object`가 클래스 템플릿인 경우 어떻게 동작할까?

```
1 struct static_member{
2     static_member() { std::cout << "static_member()" << std::endl; }
3     ~static_member() { std::cout << "~static_member()" << std::endl; }
4 };
5
6 struct instance_member{
7     instance_member() { std::cout << "instance_member()" << std::endl; }
8     ~instance_member() { std::cout << "~instance_member()" << std::endl; }
9 };
10
11 template<typename T>
12 struct Object{
13     instance_member m1;
14     static static_member m2;
15 };
16 template<typename T> static_member Object<T>::m2;
17
18 int main() {
19     std::cout << "-----" << std::endl;
20     Obj<int> obj;
21     std::cout << "-----" << std::endl;
22 }
```

- 실행 결과
- -----

-
- `instance_member`
 - -----
 - `~instance_member`
 - `static`의 생성자가 안 불린다 (지연된 인스턴스화 때문)
 - `static` 멤버 데이터도 클래스 템플릿에서는 접근하지 않으면 인스턴스화되지 않는다는 것에 유의한다
-

```
1 int main() {
2     ...
3     obj.m2(); // static 멤버 함수를 사용하면 그 때 생성자가 불린다
4     ...
5 }
```

또 다른 예제를 살펴보자

```
1 template<typename T>
2 void fn(T value){
3     if( false ){
4         *value = 10; // 인스턴스화되면서 에러가 나지만 if(false)이므로 어차피 실행이 안되게 할 수 있을까?
5     }
6 }
7
8 template<typename T>
9 void fn2(T value){
10    if constexpr ( false ){
11        *value = 10;
12    }
13 }
14
15 int main() {
16     int n = 10;
17     fn(n);
18     fn2(n);
19 }
```

- 컴파일 단계에서 `if(false)`여도 실행은 안되지만 인스턴스화에는 포함된다(코드는 컴파일 에러가 난다)
- `if constexpr(false)`는 컴파일 시간에 `if`문을 판단하여 `false`이면 포함시키지 않기 때문에 인스턴스화 되지 않는다(코드 정상 컴파일)

또 다른 예제를 살펴보자

```
1 template<typename T> void fn(T value, int){ 
2     *value = 10;
3 }
4
5 template<typename T> void fn(T value, double){
6 }
7
8 int main() {
9     fn(1, 3); // 첫번째 fn이 인스턴스화되어 컴파일 에러 발생
10    fn(1, 3.4); // 두번째 fn이 인스턴스화되어 정상 컴파일
11 }
```

6.1.10 template & friend

클래스 템플릿과 `friend` 관계에 대해 살펴보자

```
1 class Point{
2     int x, y;
3     public:
4     Point(int a, int b) : x(a), y(b){}
5
6     // private 멤버 함수에 접근하기 위해 허락해달라
7     friend std::ostream& operator<<(std::ostream& os, const Point& pt);
```

```

8   };
9
10 // Point를 cout으로 출력하기 위해 출력연산자 재정의
11 std::ostream& operator<<(std::ostream& os, const Point& pt){
12     std::cout << pt.x << ", " << pt.y << std::endl;
13     return os;
14 }
15
16 int main() {
17     Point p(1,2);
18     std::cout << p << std::endl;
19 }
```

만약 Point 클래스가 클래스 템플릿이라면 어떻게 될까?

```

1 template<typename T>
2 class Point{
3     T x, y;
4     public:
5     Point(T a, T b) : x(a), y(b){}
6
7     friend std::ostream& operator<<(std::ostream& os, const Point& pt);
8 };
9
10 template<typename T>
11 std::ostream& operator<<(std::ostream& os, const Point<T>& pt){
12     std::cout << pt.x << ", " << pt.y << std::endl;
13     return os;
14 }
15
16 int main() {
17     Point<int> p(1,2);
18     std::cout << p << std::endl;
19 }
```

- 얼핏 동작할 것 같지만 에러가 발생한다

위 코드가 왜 에러가 발생하는지 쉬운 코드부터 살펴보자

```

1 template<typename T>
2 void fn(T a) { std::cout << "T" << std::endl; }
3
4 void fn(int a) { std::cout << "int" << std::endl; }
5
6 int main() {
7     fn(3);
8 }
```

- 동일한 이름의 함수와 함수 템플릿이 있을 때 **함수가 우선적으로 선택된다**
- 만약 함수 템플릿이 있는데 함수는 선언만 있다면 어떻게 될까?

```

1 template<typename T>
2 void fn(T a) { std::cout << "T" << std::endl; }
3
4 void fn(int a); // 선언만 존재함
5
6 int main() {
7     fn(3);
8 }
```

- 함수가 **선언만 있어도 함수 템플릿보다 우선적으로 선택된다**
- 위 코드는 선언만 있으므로 링크 에러가 발생한다

이를 참고하여 Point의 인스턴스화된 코드를 보자

```

1 // int로 인스턴스화된 클래스
2 template<>
3 class Point<int>{
4     int x, y;
5     public:
6     Point(int a, int b) : x(a), y(b){}
7
8     // 함수 템플릿이 아닌 일반함수 operator<<의 선언이 된다
9     friend std::ostream& operator<<(std::ostream& os, const Point<int>& pt);
10 };
11
12 template<typename T>
13 std::ostream& operator<<(std::ostream& os, const Point<T>& pt){
14     std::cout << pt.x << ", " << pt.y << std::endl;
15     return os;
16 }
17
18 int main() {
19     Point<int> p(1,2);           // 선언만 존재하는 operator<< 호출로 링크 에러 발생
20     std::cout << p << std::endl;
21 }
```

- `friend std::ostream& operator<<(std::ostream& os, const Point<int>& pt);`; 함수가 함수템플릿보다 우선적으로 선택되기 때문에
- `main`에서 `Point<int> p(1,2);`를 호출하면 `선언만 존재하는 operator<<`로 인해 링크 에러가 발생하는 것이다

```

1 template<typename T>
2 class Point{
3     T x, y;
4     public:
5     Point(T a, T b) : x(a), y(b){}
6
7     // 템플릿을 사용하여 선언해주면 함수 템플릿이 정상적으로 friend에 등록된다
8     template<typename U>
9     friend std::ostream& operator<<(std::ostream& os, const Point<U>& pt);
10 };
11
12 template<typename T>
13 std::ostream& operator<<(std::ostream& os, const Point<T>& pt){
14     std::cout << pt.x << ", " << pt.y << std::endl;
15     return os;
16 }
17
18 int main() {
19     Point<int> p(1,2);
20     std::cout << p << std::endl;
21 }
```

- `template<typename U>`로 감싸주면 함수 템플릿에 대한 `friend` 구문이 되어 컴파일이 정상적으로 된다
- 하지만 `int`로 인스턴스화될 때 `Point<int>`가 `operator<<(Point<U>)`와 `friend` 관계가 된다
- 코드의 의도는 오직 `Point<int>`가 `operator<<(Point<int>)`와 `friend` 관계이길 원한다

```

1 template<typename T>
2 class Point{
3     T x, y;
4     public:
5     Point(T a, T b) : x(a), y(b){}
6
7     // 최종적으로 템플릿 코드를 지우고 int에 대한 코드 구현을 작성해준다
8     friend std::ostream& operator<<(std::ostream& os, const Point& pt) {
9         std::cout << pt.x << ", " << pt.y << std::endl;
10        return os;
11    }
```

```

12     };
13
14     template<typename T>
15     std::ostream& operator<<(std::ostream& os, const Point<T>& pt){
16         std::cout << pt.x << ", " << pt.y << std::endl;
17         return os;
18     }
19
20     int main() {
21         Point<int> p(1,2);
22         std::cout << p << std::endl;
23     }

```

- 최종적으로 템플릿 코드를 지우고 **int**에 대한 코드 구현을 작성해준다
- **Point<int>**와 **operator<<(Point<int>)**가 **friend** 상태인 이상적인 코드가 완성되었다

6.1.11 variable template, using template

c++ 템플릿의 한 종류인 변수 템플릿(variable template)에 대해 살펴보자

```

1 constexpr double pi = 3.141592;
2
3 int main() {
4     double area1 = 3*3*pi;
5 }

```

- **pi**는 상수이다

템플릿을 사용해서 만들 수도 있다

```

1 template<typename T>
2 constexpr T pi = static_cast<T>(3.141592);
3
4 int main() {
5     double area1 = 3*3*pi<double>;
6     float area1 = 3*3*pi<float>;

```

- 변수 템플릿을 사용하여 위와 같이 타입별 상수를 만들 수도 있다
- 변수 템플릿은 왜, 언제 사용하는가?
- 주로 **template specialization 문법**과 같이 사용한다
- **type_traits** 구현의 핵심이 되는 문법이다
- 실제 STL의 구현에서 다양하게 활용되고 있다

변수 템플릿 이외에도 템플릿의 한 종류인 **using template**에 대해 알아보자

```

1 typedef std::unordered_set<int> SET; // c style
2 using SET = std::unordered_set<int>; // c++ style (c+11)
3
4 int main() {
5     SET s1;
6 }

```

- **std::unordered_set<int>**의 이름이 길기 때문에 alias하고 싶다
- alias를 하는 다양한 문법들이 존재한다 (**typedef**, **using**)
- **std::unordered_set**의 **int**, **double**, **long**, **char** 버전을 같이 사용하고 싶을 때 일일히 alias를 지정해줘야 할까?
- 타입이 아닌 템플릿에 대한 별명을 만들 수는 없을까?

- 타입 alias: **std::unordered_set<int> --> SET**
- 템플릿 alias: **std::unordered_set --> SET (SET<int> -> std::unordered_set<int>)**

```

1 template<typename T>

```

```

2   using SET = std::unordered_set<T>; 템플릿 alias
3
4   int main() {
5       SET<int> s1;
6   }

```

using template을 사용하는 다른 예제를 살펴보자

```

1   int main() {
2       // pq는 일반적으로 인자를 하나만 사용하지만
3       std::priority_queue<int> pq1;
4
5       // 코딩을 하다보니 세번째 인자를 쓸 일이 많다고 해보자
6       std::priority_queue<int, std::vector<int>, std::less<int>> pq2;
7       std::priority_queue<int, std::vector<int>, std::greater<int>> pq3;
8   }

```

- 두번쩨 std::vector<int>를 안쓰고 첫번째, 세번째 인자만 쓸수는 없을까?

```

1   template<typename T, typename Pred>
2   using pqueue = std::priority_queue<T, std::vector<T>, Pred>;
3
4   int main() {
5       pqueue<int, std::greater<int>> pq4; // 첫번째, 세번째 인자만 사용 가능!
6   }

```

6.2 More than basic

6.2.1 dependent name

```

1   struct Object {
2       using type = int;
3       static constexpr int value = 10;
4       template<typename T> struct rebind {
5   };
6   };
7
8   template<typename T>
9   void foo(T obj) {
10      Object::value * 10;    // 10*10
11      Object::type * p1;    // int* p
12      Object::rebind<int> a;
13  }
14
15  template<typename T>
16  void goo(T obj) {
17      T::value * 10;        // error!
18      T::type * p1;        // error!
19      T::rebind<int> a; // error!
20  }
21
22  int main() {
23     Object obj;
24     foo(obj);
25     goo(obj);
26  }

```

- Object 클래스의 멤버 함수는 모두 Object::xxx를 통해 접근해야 한다
- foo는 정상적으로 컴파일되지만 goo는 컴파일 에러가 발생한다
- value, type, rebind가 값인지 타입인지 T::xxx만 봐서는 모르기 때문이다
- value, type, rebind와 같은 것들을 dependent name이라고 부른다
- **dependent name** : 논타입(non-type), 타입(type), 템플릿(template) 3가지 종류가 있다

-
- dependent name의 타입에는 `typename`을 붙여야 하고 템플릿에는 `template`을 붙여야 한다

```
1 template<typename T>
2 void goo(T obj) {
3     T::value * 10;
4     typename T::type * p1;      // ok
5     typename T::template rebind<int> a; // ok
6 }
```

또 다른 예제를 살펴보자

```
1 template<typename T> struct Object {
2     using type = int;
3
4     template<typename U>
5     void mf() { }
6 };
7
8 template<typename T>
9 void foo() {
10     Object<int>::type t1; // ok 타입이 특정되어 있으므로 컴파일 성공
11     typename Object<T>::type t2; // ok
12
13     Object<int> obj1;
14     obj1.mf<int>(); // ok
15
16     Object<T> obj2;
17     obj2.template mf<int>(); // ok
18 }
19
20 int main() {
21     foo<int>();
22 }
```

- 이번에는 `Object` 구조체 자체도 템플릿으로 작성되어 있다
- 하지만 `Object<T>::type t2`과 같이 선언하면 컴파일 에러가 발생한다
- 이를 해결하기 위해 `typename Object<T>::type t2`과 같이 작성해줘야 한다
- 그리고 `Object<T> obj2;` 이후 `obj2.mf<int>()`를 수행하면 컴파일 에러가 발생한다
- 이를 해결하기 위해 `obj2.template mf<int>()`과 같이 입력해줘야 한다

dependent name 관련하여 `value_type`이라는 개념에 대해 살펴보자

```
1 template<typename T>
2 void print_first_element(const T& v) {
3     ? n = v.front(); // 어떤 타입으로 받을 것인가?
4     std::cout << n << std::endl;
5 }
6
7 int main() {
8     std::vector<int> c = {1,2,3,4,5};
9     std::vector<double> c = {1,2,3,4,5};
10    std::list<double> c = {1,2,3,4,5};
11
12    print_first_element(c);
13 }
```

- C++11부터는 `auto`를 사용하면 된다
- `auto` 없이 하는 방법은 없을까?
- `T::value_type`을 사용하면 컨테이너의 타입을 알 수 있다

```
1 template<typename T>
2 void print_first_element(const T& v) {
3     typename T::value_type n = v.front();
```

```
4     std::cout << n << std::endl;
5 }
```

- STL의 모든 컨테이너에는 `value_type`이라는 멤버 타입이 있어서 컨테이너가 저장하는 타입이 필요할 때 사용한다
- dependent name 특성 상 앞에 `typename`을 붙여야 하는 것에 유의한다
- 실제 코딩을 할 때는 `auto`로 사용하는 것이 편리하다

6.2.2 template parameter

```
1 template<typename T> class list {
2 };
3
4 template<typename T, int N, template<typename> class C>
5 class Object {
6 };
7
8
9 int main() {
10     Object< ? > obj1;
11 }
```

- `Object`에 어떤 템플릿 키워드를 입력해야 할까?

```
1 Object<int, 10, list> obj1;
```

- 첫번째 파라미터 `T`는 타입을 입력하면 되고
- 두번째 파라미터 `N`은 정수를 입력하면 된다
- 세번째 파라미터 `template<typename> class C`는 타입 1개를 받는 템플릿을 입력하면 된다

템플릿 파라미터(template parameter)의 종류는 다음과 같다

1. type (`typename T`)
2. non-type (`int N`)
3. template (`template<typename> class C`)

템플릿 파라미터 중 template 경우에 대해 살펴보자

```
1 template<typename T>
2 class stack{
3     std::vector<T> c;
4 public:
5     void push(const T& value) { c.push_back(value); }
6     void pop() { c.pop_back(); }
7     T& top() { return c.back(); }
8 }
9
10 int main(){
11     stack<int> s1;
12     s1.push(10);
13 }
```

- `stack` 클래스는 내부적으로 `vector`를 사용하고 있다
- 반드시 `vector`를 사용할 필요 없이 사용자가 결정할 수 있도록 변경해보자

```
1 template<typename T, typename C = std::deque<T> >
2 class stack {
3     C c;
4 public:
5     ...
6 };
7
```

```
8     int main() {
9         stack<int, std::list<int>> s1;
10        stack<int, std::vector<int>> s2;
11        stack<int> s3;
12    }
```

- `typename` C를 추가하여 컨테이너를 변경할 수 있으며 변경하지 않을 경우 기본값을 선택할 수 있다(코드에서는 `deque<T>`)
- 위 코드는 STL에서 스택의 원리와 동일하다
- `std::stack`: 선형 컨테이너(`vector`, `list`, `deque`)의 멤버 함수 이름을 스택처럼 사용할 수 있게 변경하였다(이를 container adapter라고 부르며 어댑터 패턴의 한 종류이다)

STL의 `std::stack`은 `stack<int, std::vector<int>>`와 같이 템플릿 인자를 받는데 이를 `stack<int, std::vector>` 와 같이 타입을 생략할 수는 없을까?

```
1 template<typename T, template<typename, typename> class C >
2 class stack {
3     C< T, std::allocator<T> > c;
4     public:
5     ...
6 };
7
8 int main() {
9     stack<int, std::vector> s1;
10 }
```

- 첫번째 인자 T에 타입을 입력하고
- 두번째 인자 `class C`에는 인자가 2개짜리인 템플릿을 넘겨주는 형식으로 변경하였다

템플릿 파라미터 `class C`의 두번째 인자를 디폴트로 받고 싶으면 다음과 같이 작성하면 된다.

```
1 template<typename T,
2         template<typename, typename> class C,
3         typename Ax = std::allocator<T> >
4 class stack {
5     C< T, Ax > c;
6     public:
7     ...
8 };
```

`class C`를 넘겨주지 않아도 디폴트 값으로 동작하게끔 만들고 싶으면 다음 코드를 추가해준다.

```
1 template<typename T,
2         template<typename, typename> class C = std::deque,
3         typename Ax = std::allocator<T> >
4 class stack {
5     C< T, Ax > c;
6     public:
7     ...
8 };
```

- 이렇게 템플릿 인자로 템플릿을 넘겨주는 방법을 **template template parameter**라고 한다

만약 사용자가 직접 작성한 `List` 클래스를 사용하고 싶다고 해보자.

```
1 template<typename T> class List {
2     T temp = 0;
3     public:
4     ...
5 };
6
7 template<typename T,
8         template<typename, typename> class C = std::deque,
9         typename Ax = std::allocator<T> >
10 class stack{
```

```

11     C< T, Ax > c;
12     public:
13     ...
14 }
15
16 int main() {
17     stack<int, List > s1; // error!
18 }
```

- List는 템플릿 인자가 1개이므로 템플릿 인자 2개를 받는 `class C`에 들어가지 못한다
- 하지만 `std::deque`는 두번째 템플릿 인자가 디폴트 값이 존재하기 때문에 템플릿 인자를 1개만 적어도 사용할 수 있다

```

1 template<typename T> class List {
2     T temp = 0;
3     public:
4     ...
5 };
6
7 template<typename T,
8         template<typename> class C = std::deque >
9 class stack{
10     C<T> c;
11     public:
12     ...
13 }
14
15 int main() {
16     stack<int, List > s1; // ok!
17 }
```

- 두번째 파라미터를 입력할 수 없기 때문에 `Ax`는 제거하였다

템플릿 파라미터 중 non-type인 것들에 대해 살펴보자

```

1 enum Color1 { red, blue, green };
2 enum class Color2 { red, blue, green };
3
4 void foo(int a) {}
5
6 template<int N, double d,
7           Color1 C1, Color2 C2,
8           int* P, void(*FP)(int)>
9 class NTTP {
10     ...
11 };
12
13 int main() {
14     int n = 10;
15     static int s = 0;
16     NTTP< ? > t; // 어떤 파라미터를 넘겨야 할까?
17 }
```

non-type 파라미터들을 흔히 NTTP(non-type template parameter)라고 한다

- 정수형 상수 (변수는 안됨)
- 실수형 상수 (c++20부터)
- `enum` 상수 (`enum`, `enum class` 가능)
- 포인터, 함수 포인터 (`static` 변수만 가능, 지역변수 주소 안됨)
- `auto` (c++17부터)

```

1 void foo(int a) {}
2
3 int main() {
4     int n = 10;
5     static int s = 0;
6
7     NTTP< 10, 3.4, Color1::red, Color2::red, s, foo > t;
8 }
```

- `int*` 템플릿에 대하여 지역변수 `n`의 주소는 넘길 수 없다
- 대부분은 거의 사용되지 않으나 정수형 상수가 많이 사용된다

NTTP 중 `auto`에 대해 살펴보자

```

1 template<int N, double D, auto A>
2 struct Triple {
3     Triple() {
4         std::cout << N << std::endl;
5         std::cout << D << std::endl;
6         std::cout << A << std::endl;
7     }
8 };
9
10 int main(){
11     static int n = 0;
12     Triple< 10, 3.4, 10 > t1;
13     Triple< 10, 3.4, 3.4 > t2;
14     Triple< 10, 3.4, n > t3;
15 }
```

- `auto A`에는 non-type 파라미터 중 아무거나 전달해도 된다

함수 파라미터의 `auto`와 템플릿 파라미터 `auto`의 차이를 헷갈리기 쉬운데 아래 코드를 보자

```

1 template<auto A>
2 struct Object {
3 };
4
5 void foo(auto a) {
6 }
```

- 템플릿 파라미터 `auto` (c++17) : `int`, `double` 등 모든 non-type 파라미터가 전달 가능하다
- 함수 파라미터 `auto` (c++20) : 템플릿 프로그래밍을 축약한 형태이다. 즉 `template<typename T> void foo(T a)` 와 동일하다

non-type 파라미터를 사용하는 간단한 예제를 살펴보자

```

1 int main() {
2     int x[5] = {1,2,3,4,5};
3
4     std::vector<int> v = {1,2,3,4,5};
5     std::array<int, 5> a = {1,2,3,4,5};
6 }
```

- 공통점 : 모든 요소를 연속된 메모리에 보관하고 [] 연산자를 사용해서 요소에 접근할 수 있다

	요소 저장 공간	멤버 함수	크기 변경
raw array	stack	X	X
<code>std::vector</code>	heap	O	O
<code>std::array</code>	stack	O	X

따라서 `std::array`는 실제 배열을 사용하기 편리하게 만든 컨테이너라고 볼 수 있다. `std::array` 클래스를 직접 만들어보자.

```

1 template<typename T, std::size_t N>
2 struct array {
3     T arr[N];
4
5     constexpr std::size_t size() const { return N; }
6     T& operator[](int idx) { return arr[idx]; }
7     const T& operator[](int idx) const { return arr[idx]; }
8
9     using value_type = T;
10    using iterator = T*;
11
12    auto begin() { return arr; }
13    auto end() { return arr+N; }
14 };
15
16 int main(){
17     array<int, 5> a = {1,2,3,4,5};
18
19     std::cout << a.size() << std::endl;
20     a[0] = 10;
21
22     for(auto e : a) {
23         std::cout << e << std::endl;
24     }
25 }
```

- `std::array` 컨테이너를 만들 때 non-type 파라미터가 사용된다

템플릿의 디폴트 파라미터(default parameter)에 대해 살펴보자

```

1 template<typename T1 = char, typename T2 = float>
2 struct Object {
3     Object() {
4         std::cout << typeid(T1).name() << std::endl;
5         std::cout << typeid(T2).name() << std::endl;
6     }
7 };
8
9 int main() {
10     Object<int, double> obj1; // int double
11     Object<int> obj2; // int float
12     Object<> obj3; // char float
13
14     Object obj4; // char float (c++17부터 괄호 생략 가능)
15 }
```

STL을 사용한 예제 코드도 살펴보자

```

1 int main() {
2     std::vector<int> v = {1,2,3,4,5};
3
4     std::sort(v.begin(), v.end(), std::less<int>());
5     std::sort(v.begin(), v.end(), std::less()); // c++17부터 생략 가능
6     std::sort(v.begin(), v.end(), std::less{}); // c++17부터 생략, 중괄호 초기화 가능
7 }
```

6.2.3 Template type deduction

```

1 template<typename T>
2 void fn(T arg) {
3     ...
4 }
```

```

6   int main() {
7     int n = 10;
8     double d = 3.4;
9     const int c = 10;
10
11    fn<int>(3); // T = int
12
13    // 타입 인자를 생략하면?
14    fn(n); // T = int
15    fn(d); // T = double
16    fn(c); // T = const int가 될 것인가?
17 }

```

- 템플릿 인자 추론(template type deduction)은 사용자가 타입 파라미터를 생략할 경우 컴파일러가 [함수 인자를 보고 타입을 결정](#)하는 것을 말한다
- fn(c)의 경우 `const int`가 아닌 `int`로 추론한다는 점을 유의해야 한다(추후 설명)

추론된 타입을 확인하기 위해 인스턴스화 된 함수 이름을 출력할 수 있는데 이를 위한 매크로는 다음과 같다

```

1 #if defined(__GNUC__)
2   #define _FNAME_ __PRETTY_FUNCTION__
3 #elif defined(_MSC_VER)
4   #define _FNAME_ __FUNCSIG__
5 #else
6   #define _FNAME_ __FUNCTION__
7 #endif

```

```

1 template<typename T>
2 void f1(T arg) {
3   std::cout << _FNAME_ << std::endl;
4 }
5
6 template<typename T>
7 void f2(T& arg) {
8   std::cout << _FNAME_ << std::endl;
9 }
10
11 template<typename T>
12 void f3(T&& arg) {
13   std::cout << _FNAME_ << std::endl;
14 }
15
16 int main() {
17   int n = 10;
18   int& r = n;
19   const int c = 10;
20   const int& cr = c;
21
22   f1(n);      // T = int
23   f1(c);      // T = int
24   f1(r);      // T = int
25   f1(cr);    // T = int
26
27   f2(n);      // T = int,           arg = int&
28   f2(c);      // T = const int, arg = const int&
29   f2(r);      // T = int,           arg = int&
30   f2(cr);    // T = const int, arg = const int&
31
32   f3(3);      // T = int,           arg = int&&
33   f3(n);      // T = int&,        arg = int&
34   f3(c);      // T = const int&, arg = const int&
35   f3(r);      // T = int&,        arg = int&
36   f3(cr);    // T = const int&, arg = const int&
37 }

```

-
- template type deduction은 함수 인자 모양에 따라 3가지 규칙이 있다
 - 1) T : 함수 인자가 가진 "const, volatile, reference" 속성을 제거하고 T 타입을 결정한다
 - 2) T& : 함수 인자가 가진 "reference" 속성만 제거하고 T 타입을 결정. const, volatile은 유지
 - 3) T&& : 아래 테이블 같이 perfect forwarding이 적용됨

T&&는 forwarding reference라고 불리며 lvalue, rvalue를 모두 받을 수 있는 템플릿이다

	T	T&&
3 (rvalue)	int	int&&
n (lvalue)	int&	int&

다음으로 auto의 타입 추론 규칙에 대해 살펴보자. 이는 템플릿의 타입 추론 규칙과 동일하다.

T arg = 함수인자	함수의 인자를 보고 T의 타입 결정
auto n = 우변	우변을 보고 auto의 타입 결정

```

1 template<typename T>
2 void fn(T arg) {
3
4 }
5
6 int main() {
7     const int c = 10;
8     fn(c);           // c를 보고 타입 추론
9             // T arg = fn(c)
10
11    // auto는 우변을 보고 타입 추론
12    auto a1 = c;    // auto = int
13    auto& a2 = c;  // auto = const int, a2 = const int&
14
15    int n = 10;
16    auto&& a3 = 3; // auto = int, a3 = int&&
17    auto&& a4 = n; // auto = int&, a4 = int&
18 }
```

다음으로 argument decay라는 방법이 대해 살펴보자

```

1 template<typename T> void f1(T arg){
2     std::cout << _FNAME_ << std::endl;
3 }
4
5 template<typename T> void f2(T& arg){
6     std::cout << _FNAME_ << std::endl;
7 }
8
9 int main() {
10    int x[3] = {1,2,3};
11    f1(x);
12    f2(x);
13 }
```

- f1(x), f2(x)는 각각 어떤 타입으로 x를 받을까?
- 템플릿의 추론 규칙은 auto의 추론 규칙과 동일하므로 다음 코드를 보고 힌트를 얻자

```

1 int main() {
2     int x[3] = {1,2,3};
3
4     auto a1 = x;    // auto = int*, int* a1 = x;
5     auto& a2 = x; // auto = int[3]
6             // a2 = int(&)[3], a2가 int[3]을 가르키는 참조라는 뜻
7 }
```

- x의 정확한 타입은 `int[3]`이다
- `auto a1 = x`를 정확히 `int a1[3] = x`로 타입 추론하면 컴파일 에러가 발생한다(배열을 만들 때 다른 배열의 이름으로 초기화 되지 않으므로). 따라서 `auto = int*`과 같이 초기화 된다
- `auto& a2 = x`는 배열 x를 가르키는 참조가 되므로 `int(&a2)[3] = x`와 같이 된다(컴파일 정상적으로 됨)
- 즉, `auto`로 가리키면 포인터가 되고 `auto&` 참조로 가리키면 배열을 유지한다

```

1 template<typename T> void f1(T arg){
2     std::cout << _FNAME_ << std::endl;
3 }
4
5 template<typename T> void f2(T& arg){
6     std::cout << _FNAME_ << std::endl;
7 }
8
9 int main() {
10     int x[3] = {1,2,3};
11     f1(x);           // T = int*, arg= int*
12     f2(x);           // T = int[3], arg= int(&) [3]
13 }
```

- 템플릿 추론 규칙은 `auto` 추론 규칙과 동일하므로 위와 같이 추론된다
- `f1(x)`, `auto a1=x;` 같이 배열을 전달했을 때 포인터로 받는 현상을 **argument decay**라고 한다

argument decay 관련된 다른 예제를 살펴보자

```

1 template<typename T>
2 void f1(T s1, T s2) {
3 }
4
5 template<typename T>
6 void f2(const T& s1, const T& s2) {
7 }
8
9 int main() {
10     f1("banana", "apple"); // ok
11     f2("banana", "apple"); // error
12 }
```

- `f1`은 에러가 없는데 `f2`에서는 왜 에러가 발생할까?
- 문자열의 정확한 타입은 `char[]`이다 (`char*`가 아님)
- `f1`은 argument decay에 의해 `f1(char[7], char[6])` → `f1(char*, char*)`가 되어 정상적으로 컴파일된다
- 반면에 `f2`는 참조로 받으므로 `f2(char[7], char[6])`가 그대로 전달되어 둘의 타입이 다르므로 같은 `T`로 받지 못해 에러가 발생한다

6.2.4 Class template deduction guide

클래스 템플릿 타입 추론 가이드(class template type deduction guide)라는 문법에 대해 살펴보자

```

1 template<typename T> T square(T a) {
2     return a*a;
3 }
4
5 template<typename T, typename U> struct PAIR {
6     T first;
7     U second;
8
9     PAIR() = default;
10    PAIR(const T& a, const U& b) : first(a), second(b) {}
11 }
12
13 int main() {
14     square<int>(3);    // ok
15     square(3);         // ok
16 }
```

```
16
17     PAIR<int, double> p1(3, 3.4); // ok
18     PAIR p2(3, 3.4);           // ?
19 }
```

-
- `square(3)`는 타입을 명시 안 해줘도 파라미터 타입을 추론하여 컴파일이 된다(c++17)
 - 그렇다면 클래스 템플릿도 타입을 생략하여 초기화할 수 있을까?
 - 함수 템플릿: 타입 파라미터를 전달하지 않으면 함수 인자를 통해 타입을 추론한다
 - 클래스 템플릿: c++14까지는 타입 추론이 안되었으나 c++17부터 타입 추론이 가능하다

```
1 // class template type deduction guide
2 PAIR() -> PAIR<int, int>;
3
4
5 int main() {
6     PAIR p3;
7 }
```

-
- 디폴트 생성자도 잘 동작할까?
 - `PAIR()` = `default`에서 템플릿에 대한 아무런 근거가 없기 때문에 컴파일 에러가 발생한다
 - 하지만 **class template type deduction guide** 문법을 사용하면 정상적으로 컴파일이 된다
 - `PAIR() -> PAIR<int, int>;` : 사용자가 디폴트 생성자를 사용했을 때 `PAIR<int, int>` 생성자를 호출하라

deduction guide 문법에 대해 조금 더 살펴보자

```
1 template<typename T> class List {
2     public:
3         List() = default;
4         List(std::initializer_list<T> e) {}
5
6         template<typename C>
7         List(const C& c) {}
8     };
9
10    int main() {
11        std::vector<int> v = {1,2,3,4};
12
13        List<int> s1; // ok
14        List s2;      // error! 타입을 추론할 방법이 없음
15        List s3 = {1,2,3,4}; // ok
16        List s4(v);      // error!
17    }
```

-
- `List s3 = {1,2,3,4};`에서 배열이 `int`라는 것을 컴파일러가 알기 때문에 컴파일이 정상적으로 된다
 - `List s4(v);`는 `C`가 벡터라는 것을 알지만 `T`에 대한 정보는 없기 때문에 컴파일 에러가 발생한다

```
1 template<typename C>
2 List(const C&) -> List< typename C::value_type >;
3
4 int main() {
5     ...
6     List s4(v); // ok
7 }
```

-
- 위 코드를 추가하면 `C`의 요소의 타입으로 `T`를 추론하기 때문에 컴파일이 정상적으로 수행된다
 - `List s5(v.begin(), v.end());`와 같이 받을 수도 있을까?

```
1 template<typename T> class List {
2     ...
3     template<typename IT>
4     List(IT first, IT last) {}
5 }
```

```

6
7     template<typename IT>
8     List(IT first, IT last) -> List< typename IT::value_type >;
9
10    int main() {
11        ...
12        List s5(v.begin(), v.end()); // ok
13    }

```

- iterator 또한 value_type이 존재하기 때문에 추론 가능하다

6.2.5 object generator

object generator라는 기술에 대해 살펴보자

```

1 void foo { std::cout << "foo" << std::endl; }

2
3     template<typename T> class scope_exit {
4         T func;
5     public:
6         scope_exit(const T& f) : func(f) {}
7         ~scope_exit() { func(); }
8     };
9
10    int main() {
11        scope_exit ce1(&foo);
12        std::cout << "-----" << std::endl;
13    }

```

- scope_exit 객체가 foo 함수를 등록하고 소멸될 때 함수를 호출한다 (----- 다음에 호출됨)
- c++17부터는 scope_exit 타입 인자를 생략할 수 있지만 c++14에서는 scope_exit<void(*)()> ce1(&foo); 같이 타입 인자를 명시해줘야 한다
- 만약 foo의 모양이 복잡하다면 타입 인자를 어떻게 넘겨줄 수 있을까?(c++17 이후에는 상관없음)
- **object generator** : 클래스 템플릿의 타입 인자가 복잡할 경우 사용했던 기술
- 클래스 템플릿은 타입 추론 될 수 있지만(c++17 이전), 함수 템플릿은 타입 추론 될 수 있다는 문법을 활용한 기술이다

object generator 기술을 살펴보자. 우선 scope_exit<T>를 반환하는 함수를 만든다

```

1 template<typename T>
2     scope_exit<T> make_scope_exit(const T& f){
3         return scope_exit<T>(f);
4     }
5
6     int main() {
7         // scope_exit<void(*)()> ce1(&foo);
8         auto ce1 = make_scope_exit(&foo);
9         std::cout << "-----" << std::endl;
10    }

```

- auto로 함수의 리턴값을 받으면 자동으로 복잡한 타입이 추론된다 (c++11 사용 가능)
- STL에 object generator 기술이 많이 적용되어 있다

```

1 template<typename T> void fn(const T& a) {}
2
3     int main() {
4         fn( std::pair<int, double>(3, 3.4) ); // original
5         fn( std::make_pair(3, 3.4) );           // object generator
6
7         fn( std::tuple<int, double, int, char>(3, 3.4, 4, 'A') );
8         fn( std::make_tuple(3, 3.4, 4, 'A') );
9     }

```

- STL의 `make_pair`, `make_tuple`를 사용하면 타입 생략이 가능하다

c++17 부터는 클래스 템플릿도 타입 생략이 가능해져서 다음과 같이 사용할 수 있다.

```
1 int main() {
2     fn( std::pair(3, 3.4) );           // ok c++17
3     fn( std::tuple(3, 3.4, 4, 'A') ); // ok c++17
4 }
```

6.2.6 `std::type_identity`

type identity라는 개념에 대해 살펴보자

```
1 template<typename T>
2 struct type_identity {
3     using type = T;
4 };
5
6 template<typename T> void f1(T arg) {}
7 template<typename T> void f2(type_identity<T>::type arg) {}
8
9 int main() {
10    f1<int>(3);      // ok
11    f1(3);          // ok
12
13    f2<int>(3);      // ok
14    f2(3);          // ?
15 }
```

- 만약 `f1` 같이 `int`를 명시하거나 생략하거나 전부 컴파일 가능하게 하고 싶지 않고 `f1<int>`일 때만 컴파일되게 하고 싶다면 어떻게 해야 할까?
- `f2`는 `type_identity`를 활용하여 타입을 생략하면 컴파일 에러가 발생하게끔 하였다
- `type_identity<int>::type`은 `int`가 나온다. 하지만 이는 클래스 템플릿이므로 별도의 가이드 문법이 없다면 추론되지 않아서 `f2(3)` 같이 타입을 명시하지 않으면 에러가 발생한다
- c++20부터는 표준에 들어와서 별도로 클래스를 만들 필요없이 `std::type_identity`를 사용하면 된다

type identity 관련 또 다른 예제를 살펴보자

```
1 template<typename T> void f1(type_identity<T>::type arg) {}
2
3 template<typename T> void f2(T arg1, type_identity<T>::type arg2) {}
4
5 template<typename T> void f3(T arg1, T arg2) {}
6
7 int main() {
8     f1(3);          // error
9     f1<int>(3);    // ok
10
11    f2(3,4);        // ok
12
13    f3(1,2.2);     // erro
14
15    f2(1, 2.2);   // ?
16 }
```

- `f2`는 컴파일러가 3을 통해 `T`를 추론할 수 있으므로 컴파일이 정상적으로 된다
- `f3`는 두 추론된 타입이 다르므로 컴파일 에러가 발생한다
- 그렇다면 만약 `f2(1,2.2)` 같이 호출하면 어떻게 될까?
- `f2`의 `T`는 `int`로 추론됐기 때문에 2.2가 아닌 2로 변환되어 들어간다 (`f2(int, int)`)

6.3 Specialization

6.3.1 Template specialization

템플릿 특수화(template specialization)이라는 개념에 대해 살펴보자

```
1 template<typename T> class Vector{
2     T* ptr;
3     std::size_t size;
4 public:
5     Vector(std::size_t sz) : size(sz) {
6         ptr = new T[sz];
7     }
8     ~Vector() { delete[] ptr; }
9 };
10
11 int main() {
12     Vector<int> v1(5);
13     Vector<double> v2(5);
14     Vector<bool> v3(5);
15 }
```

- 마지막 `Vector<bool> v3(5)`에 주목해보자
- `v3(5)`가 호출되면 5개의 `bool` 배열이 초기화 될 것이다
- 5개가 아닌 훨씬 많은 값으로 초기화가 된다면 메모리 낭비가 될 수 있다 (`bool`은 0,1만 필요하기 때문)
- 따라서 벡터를 만들 때 임의의 타입들은 기존 코드대로 진행하고 `bool`은 다른 코드로 동작하도록 작성할 수 있다

```
1 template<> class Vector<bool> {
2     int* ptr;
3     std::size_t size;
4 public:
5     Vector(std::size_t sz) : size(sz) {
6         ptr = new int[sz/32 +1]; // 메모리 효율적 코드
7     }
8     ~Vector() { delete[] ptr; }
9 };
```

- `bool` 타입이 호출되면 위 템플릿 함수를 사용하겠다는 의미이다
- **템플릿 특수화** : 특정한 타입에 대해서만 다른 구현을 사용하고 싶을 때 사용하는 기법

포인터 타입에 대해서도 템플릿 특수화를 적용할 수 있다

```
1 template<typename T> class Vector<T*>{
2     T* ptr;
3     std::size_t size;
4 public:
5     Vector(std::size_t sz) : size(sz) {
6         ptr = new T[sz];
7     }
8     ~Vector() { delete[] ptr; }
9 };
```

- 타입이 포인터가 아닌 경우(`T`)에는 일반적으로 동작하고 타입이 포인터인 경우(`T*`)는 위 코드의 동작을 실행한다 (**부분 특수화(partial specialization)**이라고 한다)
- 일반적인 템플릿 코드는 **주 템플릿(primary template)**이라고 한다

템플릿 특수화와 관련된 또 다른 예제를 살펴보자

```
1 template<typename T, typename U> struct Object{
2     static void fn() { std::cout << "T, U" << std::endl; }
3 };
4
5 int main() {
6     Object<char, double>::fn(); // T, U
7     Object<int, short> fn(); // int, short
```

```

8     Object<short*, double>::fn(); // T*, U
9     Object<float, float>::fn(); // T, T
10    Object<int, float>::fn(); // int, U
11
12    Object<long, Object<char, short>>::fn(); // A, Object<B, C>
13 }

```

- fn() 함수를 주석과 같이 출력하도록 specialization하고 싶다고 해보자

```

1 // Specialization (특수화)
2
3 // 타입이 특정된 경우
4 template<> struct Object<int, short> {
5     static void fn() { std::cout << "int, short" << std::endl; }
6 };
7
8 // 한 타입이 포인터인 경우
9 template<typename T, typename U> struct Object<T*, U> {
10     static void fn() { std::cout << "T*, U" << std::endl; }
11 };
12
13 // 두 타입이 같은 경우. U가 필요 없으므로 생략 가능하다(partial specialization의 핵심!)
14 template<typename T> struct Object<T, T> {
15     static void fn() { std::cout << "T, T" << std::endl; }
16 };
17
18 // 첫번째 인자가 고정이고 두번째 인자가 임의의 타입인 경우도 T 생략 가능
19 template<typename U> struct Object<int, U> {
20     static void fn() { std::cout << "T, T" << std::endl; }
21 };
22
23 // 첫번째 임의의 타입, 두번째 인자가 오는데 다시 2개의 임의의 타입을 요구하면 총 3개의 임의의 타입이
24 // 선언되어야 한다
25 template<typename A, typename B, typename C> struct Object<A, Object<B,C>>{
26     static void fn() { std::cout << "A, Object<B,C>" << std::endl; }

```

- 주 템플릿(primary template)의 인자가 2개일 때 **부분 특수화 수행 시 인자의 갯수는 다를 수 있다**(핵심!)
- 하지만 Object<..., ...> 부분을 작성할 때는 반드시 2개를 표기해야 한다

다음으로 템플릿 특수화와 부분 특수화를 사용할 때 유의해야 할 특징에 대해 살펴보자

```

1 template<typename T> struct remove_pointer {
2     static void print() {
3         std::cout << typeid(T).name() << std::endl;
4     }
5 };
6
7 // 부분 특수화
8 template<typename T> struct remove_pointer<T*> {
9     static void print() {
10         std::cout << typeid(T).name() << std::endl;
11     }
12 };
13
14 int main() {
15     remove_pointer<int>::print(); // int
16     remove_pointer<int*>::print(); // int
17 }

```

- 두 print 함수는 어떤 값을 출력할까?
- 두 함수 모두 int 값을 출력한다
- int* 타입이 부분 특수화를 사용할 때 T*를 호출하는데 이 때 T 자체는 int이므로 typeid(T).name() 함수는 int를 출력한다

다음으로 부분 특수화와 디폴트 파라미터의 관계에 대해 알아보자

```
1 template<typename T1, typename T2 = short>
2 struct Object {
3     static void print() {
4         std::cout << typeid(T1).name() << ", " << typeid(T2).name() << std::endl;
5     }
6 };
7
8 int main() {
9     Object<int*>::print(); // int*, short
10 }
```

- 두번째 파라미터는 `short`로 고정되어 `int*`, `short`를 출력한다

```
1 // 부분 특수화
2 template<typename T1, typename T2> // T2 = short로 표기하지 않는다
3 struct Object<T1*, T2> {
4     static void print() {
5         std::cout << typeid(T1).name() << ", " << typeid(T2).name() << std::endl;
6     }
7 };
8
9 int main() {
10     Object<int*>::print(); // int, short
11 }
```

- 템플릿 특수화, 부분 특수화를 만들 때는 **디폴트 파라미터는 표기하지 않아야 한다**
- `T2 = short`로 표기하지 않아도 주 템플릿에 있는 디폴트 파라미터인 `short`로 인식된다

조금 어려운 예제를 보자

```
1 // primary template
2 template<typename T1, typename T2 = T1>
3 struct Object {
4     static void print() {
5         std::cout << typeid(T1).name() << ", " << typeid(T2).name() << std::endl;
6     }
7 };
8
9 // partial specialization
10 template<typename T1, typename T2>
11 struct Object<T1*, T2> {
12     static void print() {
13         std::cout << typeid(T1).name() << ", " << typeid(T2).name() << std::endl;
14     }
15 };
16
17 int main() {
18     Object<int*>::print(); // int, int*
19 }
```

- `T1`은 앞서 설명한 것처럼 `int`가 나와야 한다
- `T2`는 주 템플릿의 `T1`과 동일한 것으로 인식되기 때문에 `int*`로 출력된다

또 다른 예제를 살펴보자

```
1 template<typename T, int N>
2 struct Object; // 선언만 제공
3
4 template<typename T, int N>
5 struct Object<T, 1> {
6 };
```

```

7   template<typename T, int N>
8     struct Object<T, 2> {
9   };
10
11
12   int main() {
13     Object<int, 1> obj1;
14     Object<int, 2> obj2;
15     Object<int, 3> obj3; // error
16 }
```

- `int N`의 값이 1,2,3일 때 다르게 처리하고 싶다고 가정해보자
- 부분 특수화를 통해 `<T,1>`, `<T,2>` 템플릿 코드를 작성한다
- 다음으로 1,2일 때 제외하고 나머지 코드에 대해서는 동작하고 싶지 않게 하고 싶다고 가정해보자
- 주 템플릿(primary template)을 선언만하고 구현을 하지 않으면 1,2일 때만 작동하고 나머지는 작동하지 않게 된다

6.3.2 std::conditional

부분 특수화의 예제로 `std::conditional`을 직접 구현해보자

```

1 #include <type_traits>
2 int main() {
3   std::conditional<true, int, double>::type v1;
4   std::conditional<false, int, double>::type v2;
5
6   std::cout << typeid(v1).name() << std::endl; // int
7   std::cout << typeid(v2).name() << std::endl; // double
8 }
```

- `std::conditional<bool, Type1, Type2>::type` : 조건에 따라 타입을 선택하는 템플릿 (`<type_traits>` 헤더 필요)
- 첫번째 인자(`bool`)에 따라 타입 선택 (`true = Type1, false = Type2`)

`std::conditional`을 직접 구현해보자

```

1 // primary template
2 template<bool, typename T1, typename T2> struct conditional {
3   using type = T1;
4 };
5
6 // partial specialization
7 template<typename T1, typename T2> struct conditional<false, T1, T2> {
8   using type = T2;
9 };
```

또는 주 템플릿은 선언만 하고 다음과 같이 작성도 가능하다

```

1 // primary template
2 template<bool, typename T1, typename T2> struct conditional;
3
4 // partial specialization
5 template<typename T1, typename T2> struct conditional<true, T1, T2> {
6   using type = T1;
7 };
8
9 // partial specialization
10 template<typename T1, typename T2> struct conditional<false, T1, T2> {
11   using type = T2;
12 };
```

다음으로 `conditional`을 활용하는 예제를 살펴보자

```

1 // 32bit를 관리하는 클래스
2 template<std::size_t N> struct Bit {
```

```

3     std::conditional< (N>16), unsigned int, unsigned short> data;
4 };
5
6 int main() {
7     Bit<16> bit;
8     std::cout << sizeof(bit) << std::endl;
9
10    Bit<32> bit2;
11    std::cout << sizeof(bit2) << std::endl;
12 }
```

- 몇 bit을 선택하느냐에 따라 내부의 data의 타입을 다르게 하고 싶을 경우 conditional을 사용할 수 있다

6.3.3 Template meta programming

템플릿 메타 프로그래밍(template meta programming)이라는 개념에 대해 살펴보자

```

1 // primary template
2 template<std::size_t N> struct Factorial {
3     enum { value = N * Factorial<N-1>::value };
4 };
5
6 // template specialization
7 template<> struct Factorial<1> {
8     enum { value = 1 };
9 };
10
11 int main() {
12     int ret = Factorial<5>::value;
13             // 5 * F<4>::value
14             // 4 * F<3>::value
15             // 3 * F<2>::value
16             // 2 * F<1>::value
17             //      (1)
18
19     std::cout << ret << std::endl; // 120
20 }
```

- 템플릿 코드는 컴파일 시간에 계산되기 때문에 컴파일이 되고 나면 값으로 변환된다
- 이렇게 컴파일 시간에 연산을 수행하는 기법을 **템플릿 메타 프로그래밍**이라고 한다
- 일반적으로 재귀를 사용하며 재귀의 종료를 위해 템플릿 특수화 기법이 사용된다
- c++98 시절부터 사용되는 방법이었으나 c++11부터는 **constexpr** 문법을 사용하는 것이 가능하므로 현재는 아래 방법을 더 많이 사용한다

```

1 constexpr std::size_t factorial(std::size_t n) {
2     return n == 1 ? 1 : n * factorial(n-1);
3 }
4
5 int main() {
6     constexpr std::size_t ret = factorial(5);
7     std::cout << ret << std::endl; // 120
8 }
```

- **constexpr** : factorial의 파라미터가 혹시 컴파일 시간에 값을 알 수 있는 상수라면 미리 컴파일 시간에 계산한다

6.3.4 Variable template specialization

변수 템플릿의 특수화(variable template specialization)라는 기술에 대해 살펴보자

```

1 template<typename>
2 inline constexpr int made_year = -1;
3
```

```

4 class AAA {};
5 template<>
6 inline constexpr int made_year<AAA> = 2020;
7
8 class BBB {};
9 template<>
10 inline constexpr int made_year<BBB> = 2022;
11
12 int main() {
13     std::cout << made_year<AAA> << std::endl; // 2020
14     std::cout << made_year<BBB> << std::endl; // 2022
15     std::cout << made_year<int> << std::endl; // -1
16 }
```

- 업데이트가 자주 발생되는 프로그램을 작성하는데 [클래스가 추가된 연도를 관리하고 싶다고](#) 가정해보자
- 템플릿 특수화를 사용하면 각 클래스를 만든 사람이 자신이 언제 만들었는지 표기할 수 있다

c++ 표준에서 사용되고 있는 변수 템플릿 특수화를 살펴보자

```

1 int main() {
2     std::string s1 = "to be or not to be";
3     std::string s2 = s1;           // deep copy
4
5     std::string_view sv = s1; // reference
6
7     std::cout << std::ranges::enable_borrowed_range<std::string> << std::endl; // 0
8     std::cout << std::ranges::enable_borrowed_range<std::string_view> << std::endl; // 1
9 }
```

- borrowed range(빌린 range) : 자원을 소유하지 않고 다른 range(=container)가 소유한 자원을 사용하는 range
- `string_view`와 같이 동작하는 클래스들의 집합(c++20)
- borrowed range 여부를 조사하기 위해서는 `std::ranges::enable_borrowed_range`라는 변수 템플릿으로 조사해야 한다 (<range> 헤더 필요)

`std::ranges::enable_borrowed_range`는 다음과 같이 구현되어 있다

```

1 // variable template specialization
2 template<typename>
3 inline constexpr bool enable_borrowed_range = false; // 디폴트로 모든 타입들은 borrowed range가 아니다
4
5 template<typename T, typename Traits>
6 inline constexpr bool enable_borrowed_range<std::basic_string_view<T, Traits>> = true;
7 // std::basic_string_view에 대해서만 true이다
```

6.4 type_traits

6.4.1 Type traits

type traits라는 기술에 대해 살펴보자

```

1 template<typename T>
2 void fn(const T& arg) {
3     ?
4 }
5
6 int main() {
7     int n = 0;
8     fn(n);
9     fn(&n);
10}
```

- 템플릿 코드 작성 시 타입 인자 `T`가 포인터인 경우와 그렇지 않은 경우 각각 다르게 코드를 작성하고 싶다고 하자
- 그런 경우 `T`가 포인터 타입인지 여부를 결정하는 코드가 필요한데 이러한 코드들의 특성을 조사하는 라이브러리를 `type traits`라고 한다

type traits 라이브러리

1. 타입에 대한 다양한 속성을 조사하거나 (query the properties of types)
2. 변형(transformation)된 타입을 구할 때 사용 (type modifications)
3. 1990년대 말 부터 사용되기 시작
4. boost 라이브러리에서 제공하기 시작(2000년 초)
5. c++11에서 c++ 표준으로 도입되었다 (<type_traits> 헤더)

T가 포인터인지 조사하려면 다음과 같이 하면 된다

- `std::is_pointer<T>::value` (c++11)
- `std::is_pointer_v<T>` (c++17)

T에서 포인터를 제거한 타입을 구하는 방법은 다음과 같다

- `typename std::remove_pointer<T>::type` (c++11)
- `std::remove_pointer_t<T>` (c++14)

```
1 template<typename T>
2 void fn(const T& arg) {
3     bool b1 = std::is_pointer<T>::value;
4     bool b2 = std::is_pointer_v<T>;
5
6     typename std::remove_pointer<T>::type n1;
7     typename std::remove_pointer_t<T> n2;
8 }
9
10 int main() {
11     int n = 0;
12     fn(n);
13     fn(&n);
14 }
```

is_pointer를 직접 구현해보자

```
1 // primary template
2 template<typename T> struct is_pointer {
3     enum { value = false; }
4 };
5
6 // partial specialization
7 template<typename T> struct is_pointer<T*> {
8     enum { value = true; }
9 };
10
11 template<typename T>
12 void fn(const T& arg){
13     if(is_pointer<T>::value)
14         std::cout << "pointer" << std::endl;
15     else
16         std::cout << "not pointer" << std::endl;
17 }
18
19 int main() {
20     int n = 0;
21     fn(n);
22     fn(&n);
23 }
```

-
- 실제 구현에는 `const`, `volatile`, `const volatile`에 대한 부분 특수화 코드도 구현해야 한다
 - `is_pointer<T>::value`는 `enum`인 상수값을 반환하므로 컴파일 타임에 동작하는 것을 알 수 있다(메타 함수)
 - 메타 함수(meta function): 컴파일러가 컴파일 시간에 사용하는 함수

c++11 이후부터는 `constexpr` 추가되면서 구현 코드를 `enum`이 아닌 다음과 같이 변경할 수 있다

```

1 // primary template
2 template<typename T> struct is_pointer {
3     static constexpr bool value = false;
4 };
5
6 // partial specialization
7 template<typename T> struct is_pointer<T*> {
8     static constexpr bool value = true;
9 };

```

`std::is_pointer` 외에도 `std::is_const`, `std::is_array`와 같은 다른 c++ 표준 type traits들을 살펴보자

```

1 template<typename T>
2 void fn(T& arg) {
3     std::cout << std::is_const<T>::value << std::endl;
4     std::cout << std::is_array<T>::value << std::endl;
5 }
6
7 int main() {
8     int n = 0;
9     int x[3] = {1,2,3};
10    const int c = 0;
11    const int y[3] = {1,2,3};
12
13    fn(n); // 0,0
14    fn(c); // 1,0
15    fn(x); // 0,1
16    fn(y); // 1,1
17 }

```

6.4.2 `is_pointer` & `if constexpr`

앞서 만든 `is_pointer`를 사용하는 방법에 대해 살펴보자

```

1 template<typename T> struct is_pointer { enum { value = false}; };
2 template<typename T> struct is_pointer<T*> { enum { value = true}; };
3
4 template<typename T>
5 void printv(const T& value){
6     if( is_pointer<T>::value )
7         std::cout << value << " : " << *value << std::endl;
8     else
9         std::cout << value << std::endl;
10 }
11
12 int main() {
13     int n = 10;
14     printv(&n); // ok
15     printv(n); // error!
16 }

```

- `printv` 함수 템플릿 : 인자로 전달된 변수의 값을 출력하는 함수. 만약 `인자의 값이 포인터라면 변수 값(메모리 주소), 메모리 값(*value)도 출력`한다
- `printv(n)`을 컴파일 하면 컴파일 시간에 `if(false) std::cout << ... << *value << ...`를 컴파일 해야 하는데 `n`이 포인터가 아니므로 `*value` 부분에서 컴파일 에러가 발생한다

`printv` 함수 템플릿 문제를 해결하는 방법은 4가지가 존재한다

1. `std::integral_constant(int2type)` (c++11)

-
2. `std::enable_if` (c++11)
 3. `if constexpr` (c++17)
 4. `concept` (c++20)

이 중 첫번째와 세번째 단계를 자주 사용하므로 이에 대해 살펴보자. 우선 세번째 방법을 사용해보자.

```
1 template<typename T>
2 void printv(const T& value){
3     if constexpr ( is_pointer<T>::value )
4         std::cout << value << " : " << *value << std::endl;
5     else
6         std::cout << value << std::endl;
7 }
```

- `if constexpr` : 조건이 `false`인 경우 코드가 인스턴스화 된 함수에 포함되지 않음

6.4.3 int2type

앞선 문제를 `if constexpr`을 사용하지 않고 해결하는 방법에 대해 살펴보자

```
1 template<typename T> struct is_pointer { enum { value = false}; };
2 template<typename T> struct is_pointer<T*> { enum { value = true}; };
3
4 template<typename T>
5 void pointer(const T& value) {
6     std::cout << value << " : " << *value << std::endl;
7 }
8
9 template<typename T>
10 void not_pointer(const T& value) {
11     std::cout << value << std::endl;
12 }
13
14 template<typename T>
15 void printv(const T& value){
16     if ( is_pointer<T>::value )
17         pointer(value);
18     else
19         not_pointer(value);
20 }
21
22 int main() {
23     int n = 10;
24     printv(n);
25 }
```

- 포인터인 경우와 포인터가 아닌 경우를 별도의 함수 템플릿으로 분리한다
- 사용되지 않는 함수 템플릿은 인스턴스화되지 않으므로 이 점을 활용한다
- 하지만 `if`문 사용 시 컴파일 시간에 `false`로 결정되어도 모두 사용되는 것으로 간주해서 두 함수가 모두 인스턴스화된다(결국 컴파일 에러)

```
1 template<typename T>
2 void printv_imp(const T& value, ?) {
3     std::cout << value << " : " << *value << std::endl;
4 }
5
6 template<typename T>
7 void printv_imp(const T& value, ?) {
8     std::cout << value << std::endl;
9 }
10
11 template<typename T>
```

```

12 void printv(const T& value){
13     printv_imp(value, is_pointer<T>::value );
14 }
```

- 함수 오버로딩의 특성 상 동일한 이름의 함수가 여러 개 있을 때 어떤 함수를 호출할지 결정하는 것은 컴파일 시간에 결정되므로 이 특징을 사용한다
- 따라서 두번째 인자로 `is_pointer<T>::value`를 넘겨주고자 하는데 해당 값의 리턴 값은 0,1이다
- 그런데 둘은 같은 `int`이므로 함수 오버로딩이 되지 않는다

정수 0,1을 다른 타입으로 만들기 위한 `int2type`에 대해 알아보자

```

1 template<int N>
2 struct int2type {
3     enum { value = N; }
4 };
5
6 int main() {
7     fn(0); // fn(int)
8     fn(1); // fn(int) same
9
10    int2type<0> t0;
11    int2type<1> t1; // 둘은 다른 타입이다
12
13    fn(t0);
14    fn(t1); // 둘은 다른 함수를 호출한다
15 }
```

- `fn(0)`, `fn(1)`은 같은 타입의 함수를 호출한다
- `int2type<0>`, `int2type<1>`은 `enum value` 값을 각각 0,1로 채운 서로 다른 타입이다!
- `int2type` : 컴파일 시간에 결정된 정수형 상수를 타입으로 만드는 템플릿

```

1 template<int N>
2 struct int2type {
3     enum { value = N; }
4 };
5
6 template<typename T>
7 void printv_imp(const T& value, int2type<1>) {
8     std::cout << value << " : " << *value << std::endl;
9 }
10
11 template<typename T>
12 void printv_imp(const T& value, int2type<0>) {
13     std::cout << value << std::endl;
14 }
15
16 template<typename T>
17 void printv(const T& value){
18     printv_imp(value, int2type< is_pointer<T>::value>());
19 }
```

- `printv_imp` 함수가 각각 `int2type<1>`, `int2type<0>` 일 때 다르게 오버로딩되어 있기 때문에 `is_pointer` 여부에 따라 다르게 동작한다 (컴파일 성공)
- `int2type<1> a, int2type<0> b` 같이 파라미터의 이름을 명시하지 않는게 속도적으로 더 빠르다고 한다

`int2type`의 역사를 간단하게 살펴보면

- 2000년대 초반 Andrei Alexandrescu에 의해 소개됨
- c++11을 만들 때 `std::integral_constant`로 발전됨

6.4.4 `std::integral_constant`

c++11부터 추가된 integral constant라는 개념에 대해 살펴보자

```

1 template<int N> struct int2type {
2     enum { value = N };
3 };
4
5 int main() {
6     int2type<0> t1;
7     int2type<1> t2;
8 }
```

- `int2type` : `int` 정수를 하나의 타입으로 만드는 도구
- `int` 뿐만 아니라 정수형 타입의 상수(`bool`, `char`, `short`, `long` 등)도 타입으로 만들면 좋지 않을까?

```

1 template<typename T, T N> struct integral_constant {
2     static constexpr T value = N;
3 };
4
5 int main() {
6     integral_constant<int, 0> n0;
7     integral_constant<int, 1> n1;
8     integral_constant<short, 1> n2;
9     integral_constant<bool, true> n2;
10 }
```

- `std::integral_constant`(c++11) : `int2type`의 발전된 형태. 모든 정수형 타입의 상수를 타입화 할 수 있다

`std::integral_constant`에 대해 조금 더 살펴보자

```

1 template<typename T, T N> struct integral_constant {
2     static constexpr T value = N;
3
4     using value_type = T;
5     using type = integral_constant;
6
7     constexpr operator value_type() const noexcept { return value; }
8     constexpr value_type operator()() const noexcept { return value; }
9 };
10
11 int main() {
12     integral_constant<int, 34> t1;
13
14     int n1 = t1; // t1.operator int() 호출, 34
15     int n2 = t2(); // t1.operator()() 호출, 34
16 }
```

- 실제 표준에는 타입과 변환 연산자와 함수 연산자들이 정의되어 있다 (`operator value_type()`, `operator()()`)

다음으로 `std::true_type`, `std::false_type`에 대해 살펴보자

```

1 template<typename T, T N> struct integral_constant {
2     static constexpr T value = N;
3     using value_type = T;
4     using type = integral_constant;
5     constexpr operator value_type() const noexcept { return value; }
6     constexpr value_type operator()() const noexcept { return value; }
7 };
8
9 using true_type = integral_constant<bool, true>;
10 using false_type = integral_constant<bool, false>;
11
12 int main() {
13 }
```

- `std::true_type`, `std::false_type` : `true`와 `false`를 가지고 만든 타입

-
- `true`, `false` : 참, 거짓을 나타내는 값 (같은 타입 `bool`)
 - `std::true_type`, `std::false_type` : 참, 거짓을 나타내는 타입 (다른 타입)

최근 `is_pointer` 구현은 `std::true_type`, `std::false_type`을 활용하여 구현되어 있다

```
1 template<typename T, T N> struct integral_constant {
2     static constexpr T value = N;
3     using value_type = T;
4     using type = integral_constant;
5     constexpr operator value_type() const noexcept { return value; }
6     constexpr value_type operator() const noexcept { return value; }
7 };
8
9 using true_type = integral_constant<bool, true>;
10 using false_type = integral_constant<bool, false>;
11
12 template<typename T> struct is_pointer : false_type
13 template<typename T> struct is_pointer<T*> : true_type
14
15 void fn( true_type ) { std::cout << "true" << std::endl; }
16 void fn( false_type ) { std::cout << "false" << std::endl; }
17
18 int main() {
19     fn( is_pointer<int>() ); // fn(false_type) 호출
20     fn( is_pointer<int*>() ); // fn(true_type) 호출
21 }
```

- `is_pointer<T>`가 포인터인 경우 자동으로 `value=true`가 되고 포인터가 아닌 경우 자동으로 `value=false`가 된다

`std::true_type`, `std::false_type`을 활용하여 앞서 설명한 `printv`를 `int2type` 없이 다시 만들어보자

```
1 #include <type_traits>
2
3 template<typename T>
4 void printv_imp(const T& value, std::true_type) {
5     std::cout << value << " : " << *value << std::endl;
6 }
7
8 template<typename T>
9 void printv_imp(const T& value, std::false_type) {
10     std::cout << value << std::endl;
11 }
12
13 template<typename T>
14 void printv(const T& value){
15     printv_imp(value, std::is_pointer<T>());
16 }
```

- 앞서 설명한 함수들은 `<type_traits>` 헤더에 포함되어 있다

6.4.5 `std::is_pointer` implementation

`std::is_pointer` 관련하여 몇 가지 추가적인 사항에 대해 살펴보자

```
1 namespace mystd {
2     template<typename T> struct is_pointer : std::false_type{};
3     template<typename T> struct is_pointer<T*> : std::true_type{};
4 }
5
6 int main() {
7     //namespace X = std;
8     namespace X = mystd;
9
10    std::cout << X::is_pointer<int>::value << std::endl;
11    std::cout << X::is_pointer<int*>::value << std::endl;
```

```

12     std::cout << X::is_pointer<int* const>::value << std::endl;
13     std::cout << X::is_pointer<int* volatile>::value << std::endl;
14     std::cout << X::is_pointer<int* const volatile>::value << std::endl;
15 }
```

- X = `std`이면 첫번째 항만 0이고 나머지는 전부 1가 출력된다
- X = `mystd`를 사용하면 두번째 항만 1이고 나머지는 전부 0이 출력된다

따라서 제대로 동작하도록 구현하려면 `const`, `volatile`도 전부 고려해줘야 한다

```

1 namespace mystd {
2     template<typename T> struct is_pointer : std::false_type{};
3     template<typename T> struct is_pointer<T*> : std::true_type{};
4     template<typename T> struct is_pointer<T* const> : std::true_type{};
5     template<typename T> struct is_pointer<T* volatile> : std::true_type{};
6     template<typename T> struct is_pointer<T* const volatile> : std::true_type{};
7 }
```

5개를 전부 만들지 않고 2개만 만들어서 처리하는 방법도 있다

```

1 namespace mystd {
2     namespace detail{
3         template<typename T> struct is_pointer : std::false_type{};
4         template<typename T> struct is_pointer<T*> : std::true_type{};
5     }
6     template<typename T>
7     using is_pointer = detail::is_pointer<std::remove_cv_t<T>>;
8 }
```

- `std::remove_cv_t` : `const`, `volatile` 속성을 제거한 값을 반환하므로 이전과 달리 `const`, `volatile` 키워드가 있어도 포인터이면 `true`를 반환한다

`is_pointer`를 변수 템플릿으로 간편하게 사용할 수 있다

```

1 template<typename T>
2 constexpr bool is_pointer_v = std::is_pointer<T>::value;
3
4 template<typename T>
5 void fn(const T& arg) {
6     bool b1 = is_pointer_v<T>;
7 }
8
9 int main() {
10     int n = 10;
11     fn(n);
12 }
```

- `is_pointer_v`를 변수 템플릿(variable template)이라고 한다
- C++17 이후부터 표준에도 똑같이 구현되어 있다

지금까지 `is_pointer`를 만드는 방법에 대해 정리하고 최근 많이 사용하는 다른 생성 방법을 살펴보자

```

1 // #1
2 // 1. 클래스 템플릿을 먼저 만들고
3 template<typename T> struct is_pointer : std::false_type{};
4 template<typename T> struct is_pointer<T*> : std::true_type{};
5 template<typename T> struct is_pointer<T* const> : std::true_type{};
6 template<typename T> struct is_pointer<T* volatile> : std::true_type{};
7 template<typename T> struct is_pointer<T* const volatile> : std::true_type{};
8
9 // 2. 클래스 템플릿을 사용해서 변수 템플릿을 구현
10 template<typename T>
11 constexpr bool is_pointer_v = std::is_pointer<T>::value;
12
13 // -----
```

```

14 // #2
15 // 1. 변수 템플릿을 먼저 만들고
16 template<typename T> constexpr bool is_pointer_v = false;
17 template<typename T> constexpr bool is_pointer_v<T*> = true;
18 template<typename T> constexpr bool is_pointer_v<T* const > = true;
19 template<typename T> constexpr bool is_pointer_v<T* volatile> = true;
20 template<typename T> constexpr bool is_pointer_v<T* const volatile> = true;
21
22 // 2. 변수 템플릿을 사용해서 클래스 템플릿을 구현
23 template<typename T> struct is_pointer : std::integral_constant<bool, is_pointer_v<T>>

```

T가 포인터인지 조사하는 방법

c++11	<code>std::is_pointer<T>::value</code>
c++17	<code>std::is_pointer_v<T></code>

T의 포인터 여부에 따라 다른 구현을 작성하려면

(1)	<code>if constexpr</code> 로 조사 후 작성	c++17
(2)	<code>std::true_type, std::false_type</code> 으로 함수 오버로딩 사용	c++11
(3)	<code>std::enable_if</code>	c++11
(4)	concep	c++20

6.4.6 type modification traits

type traits의 중요한 특징 중 하나인 변형된 타입(modified type)을 얻는 방법에 대해 살펴보자

```

1 template<typename T>
2 void fn(const T& value) {
3     std::remove_pointer_t<T> n;
4
5     std::cout << typeid(n).name() << std::endl;
6 }
7
8 int main() {
9     int n = 0;
10    fn(n);      // int
11    fn(&n);    // int
12 }

```

- `std::remove_pointer_t<T>` : 기존 타입에서 포인터를 제거해주는 연산

`std::remove_pointer_t<T>`를 실제로 만들어보자

```

1 // primary template
2 template<typename T> struct remove_pointer {
3     using type = T;
4 };
5
6 // partial specialization
7 template<typename T> struct remove_pointer<T*> { // int*
8     using type = T; // int
9 };
10
11 template<typename T>
12 void fn(const T& value) {
13     typename remove_pointer<T>::type n1;
14
15     std::cout << typeid(n1).name() << std::endl;
16 }
17
18 int main() {
19     int n = 0;

```

```
20     fn(&n); // int
21 }
```

- `remove_pointer`의 부분 특수화를 통해 포인터일 경우 `T*`를 받음으로써 `using type = T`는 `int`가 된다

`remove_pointer`를 조금 더 발전시켜보자

```
1 template<typename T> struct remove_pointer { using type = T; };
2 template<typename T> struct remove_pointer<T*> { using type = T; };
3 template<typename T> struct remove_pointer<T* const> { using type = T; };
4 template<typename T> struct remove_pointer<T* volatile> { using type = T; };
5 template<typename T> struct remove_pointer<T* const volatile> { using type = T; };
6
7 int main() {
8     std::cout << typeid(remove_pointer<int*>::type).name() << std::endl;
9     std::cout << typeid(remove_pointer<int* const>::type).name() << std::endl;
10    std::cout << typeid(remove_pointer<int* volatile>::type).name() << std::endl;
11    std::cout << typeid(remove_pointer<int* const volatile>::type).name() << std::endl;
12 }
```

다음으로 `remove_pointer_t` 버전에 대해 살펴보자

```
1 template<typename T>
2 using remove_pointer_t = typename remove_pointer<T>::type;
3
4 template<typename T>
5 void fn(const T& value) {
6     remove_pointer_t<T>n;
7
8     std::cout << typeid(n).name() << std::endl;
9 }
10
11 int main() {
12     int n = 0;
13     fn(&n); // int
14 }
```

- `remove_pointer_t`는 길고 복잡한 코드를 단순히 alias한 것이다(`using`을 이용하므로 `using` 템플릿이다)
- C++14부터는 `std::remove_pointer_t`를 표준에서 제공한다

6.4.7 `remove_all_pointer`

이중, 삼중 포인터도 제거할 수 있는 `remove_all_pointer`에 대해 알아보자

```
1 template<typename T> struct remove_pointer { using type = T; };
2 template<typename T> struct remove_pointer<T*> { using type = T; };
3
4 int main() {
5     std::cout << typeid(remove_pointer<int*>::type).name() << std::endl; // good
6     std::cout << typeid(remove_pointer<int**>::type).name() << std::endl; // bad
7     std::cout << typeid(remove_pointer<int***>::type).name() << std::endl; // bad
8 }
```

- 이중, 삼중 포인터 `int**`, `int***`는 포인터가 제대로 제거되지 않는다. 이럴 경우에는 어떻게 처리 해야 할까?
- 모든 포인터를 제거할 수 있는 `remove_all_pointer`를 작성해보자(C++ 표준에는 제공되지 않음)

```
1 template<typename T> struct remove_all_pointer { using type = T; };
2 template<typename T> struct remove_all_pointer<T*> {
3     using type = typename remove_all_pointer<T>::type; // 재귀를 활용하여 해결
4 };
5
6 template<typename T>
7 using remove_all_pointer_t = typename remove_all_pointer<T>::type; // 쉽게 사용할 수 있게 alias 사용
```

```

8
9 int main() {
10    std::cout << typeid(remove_pointer<int*>::type).name() << std::endl; // good
11    std::cout << typeid(remove_pointer<int**>::type).name() << std::endl; // good
12    std::cout << typeid(remove_pointer<int***>::type).name() << std::endl; // good
13 }
```

6.5 Concept

6.5.1 concept

c++20부터 추가된 컨셉(concept)이라는 문법에 대해서 살펴보자

```

1 template<typename T>
2 void foo(const T& arg) {
3     bool b= std::is_pointer_v<T>;
4     std::println("{}", b);
5 }
6
7 int main() {
8     std::vector v{1,2,3};
9     foo(v);
10 }
```

- foo 함수의 T가 포인터인지 조사하고 싶은 경우 어떻게 해야 할까?
- `std::is_pointer`, `std::is_pointer_v`를 사용하면 조사할 수 있다
- 이러한 기술을 type traits라고 부르며 c++11부터 가능하다. 대부분 템플릿 특수화(template specialization), 부분 특수화(partial specialization) 문법을 통해 구현되어 있다
- 그렇다면 T가 컨테이너인지 조사할 수 있을까? - T가 컨테이너 인지 보다 더 중요한 건 "컨테이너에 대한 정의를 어떻게 할 수 있을까?" 이다

컨셉 문법(c++20)

- 타입이 가져야 할 요구 조건의 집합(named set of requirements)
- 그리고 **요구조건의 집합을 코드로 작성하는 문법**
- 컨셉 문법의 의도는 단순한 조사가 아닌 `container`와 같은 개념을 코드로 정의하는 것에 있다 (따라서 `is_container`보다 `container`를 권장)

컨셉을 만드는 기본 모양은 다음과 같다

```

1 template<typename T>
2 concept container = requires(T c) {
3     // 객체 c가 컨테이너가 되기 위해 지켜야 하는 요구 조건
4     ...
5 };
```

- `container`가 컨셉의 이름이다

`container`라는 컨셉을 직접 만들어보자

```

1 template<typename T>
2 concept container = requires(T c) {
3     c.begin();
4     c.end();
5 };
6 // container라는 것은 begin(), end() 함수가 존재해야 한다
7
8 template<typename T>
9 void foo(const T& arg) {
10     bool b= container<T>;
11     std::println("{}", b);
12 }
13 }
```

```

14     int main() {
15         std::vector v{1,2,3};
16         foo(v);
17     }

```

- 컨셉 활용: 템플릿 작성 시 조건을 만족하는지에 따라 다른 알고리즘을 작성 가능
- 컴파일 시간에 조사하게 되므로 `if constexpr`을 사용 가능
- 그 외 다양하게 활용 가능
- 사용자가 만들지 않아도 C++ 표준에서 이미 만들어진 다양한 컨셉을 제공한다(<concepts>, <iterator>, <ranges> 헤더에 있음)
- `std::input_or_output_iterator<T>` : 입력 반복자 또는 출력 반복자가 가져야 하는 조건을 정의한 컨셉(자주 사용하는 반복자)
- C++ 표준에 `container`라는 이름의 컨셉은 없고 `std::ranges::range`라는 컨셉이 있다. 왜 이렇게 이름이 명명되었는지는 추후 설명한다

컨셉을 만드는 방법을 자세히 살펴보자

```

1 constexpr bool yes() { return true; }

2
3 template<typename T>
4 concept C1 = true;
5 // concept C1 = 1; // error

6
7 template<typename T>
8 concept C2 = yes();

9
10 template<typename T>
11 concept C3 = sizeof(T) > 4;

12
13 template<typename T>
14 concept C4 = std::is_integral_v<T>;
15

16 template<typename T>
17 concept C5 = C3<T> || C4<T>;
18

19 template<typename T>
20 concept C6 = requires(T& c) {
21     c.begin()
22 };
23

24 int main() {
25     std::println("{}", C1<int>); // true
26     std::println("{}", C2<int>); // true
27     std::println("{}", C3<int>); // false (4보다 크지 않으므로)
28     std::println("{}", C4<int>); // true (정수이므로)
29     std::println("{}", C5<int>); // true (true || false 이므로)
30     std::println("{}", C6<int>); // false
31 }

```

- 컨셉도 결국은 템플릿 인자에 대한 제약들을 정리하는 일종의 템플릿이다

c++98	function template class template
c++11	using template
c++14	variable template
c++20	concept template

따라서 컨셉을 만드는 기본 모양은 템플릿을 만드는 기본 모양과 동일하다

```

1 template<typename T>
2 concept concept_name = constraint_expression;

```

- `constraint expression`에는 컴파일 할 때 조사가 가능한 조건식을 작성해야 한다

-
- bool literal, `constexpr`(`consteval`)function, type traits, `&&`, `ll`, `!` 등이 들어갈 수 있다
 - 또는 `require expression`을 작성할 수 있다

6.5.2 requires expression

`requires` 표현법에 대해 자세히 알아보자

```
1 template<typename T>
2 concept addable = requires(T a, T b){
3     a+b;
4 };
5
6 template<typename T>
7 void test(const T& arg) {
8     std::print("{}", addable<T>); // true
9 }
10
11 int main() {
12     test(5);
13 }
```

- `requires` : 타입이 가져야 하는 제약 조건을 가진 표현식
- 컴파일 시간에 조건을 조사해서 최종적으로 `bool` 타입의 prvalue를 생성한다
- 컨셉을 만들 때 주로 사용하지만 "컴파일 시간 `bool` 값이 필요한 위치에도 사용 가능"

```
1 template<typename T>
2 concept addable = requires(T a, T b){
3     a+b;
4 };
5
6 constexpr bool ret = requires( T a, T b) { a+b; }
7
8 static_assert( addable<T>, "not addable");
9 static_assert( requires( T a, T b) { a+b; }, "not addable"); // 직접 적는 것도 가능하다
```

- 위와 같이 컨셉에 사용하지 않고도 사용 가능하다

`requires` 표현법의 다양한 모양에 대해 살펴보자

```
1 template<typename T>
2 concept C1 = requires {
3     T{};
4 };
5
6 template<typename T>
7 concept C2 = requires(T a) {
8     *a;
9 };
10
11 class AAA {
12     private:
13     AAA() {}
14 };
15
16 int main() {
17     std::println("{} , {}" , C1<int>, C1<AAA>); // t, f
18     std::println("{} , {}" , C2<int>, C2<int*>); // f, t
19 }
```

- `requires` 표현법의 기본적인 모양은 다음과 같다
- (1) `requires { requirement-seq }`
- (2) `requires(params){ requirement-seq }`
- `requirement-seq`에 들어갈 문장은 크게 네 종류이다

- simple, type, compound, nested requirements

```
1 template<typename T>
2 concept C1 = requires (T a, T b) {
3     // #1 simple
4     a+b;
5     a.f1();
6
7     // #2 type
8     typename T::value_type;
9
10    // #3 compound
11    {a+b} noexcept -> std::same_as<int>;
12
13    // #4 nested
14    requires sizeof(T) > 4;
15};
```

- compound requirement : {expression} noexcept -> return-type-requirement

compound requirement에 대해 조금 더 살펴보자

```
1 template<typename T>
2 concept pointer = std::is_pointer_v<T>;
3
4 template<typename T>
5 concept C1 = requires(T a, T b) {
6     {a+b} -> int;                                // error
7     {a+b} -> std::is_pointer_v;                  // error
8
9     {a+b} -> pointer;                            // ok 아래 코드들도 동일하게 타입 T를 명시하지 않아도 된다(a+b에서 이미
10    추론됐으므로)
11    {a.f1()} -> std::same_as<int>;           // ok
12    {a.f2()} -> std::convertible_to<bool>; //ok
13};
14
15 int main() {
16     using T = double;
17     std::println("{} {}", pointer<T>);
18     std::println("{} {}", std::same_as<T, int>);
19     std::println("{} {}", std::convertible_to<T, bool>);
20 }
```

- compound의 return type requirement는 타입을 직접 표기하거나 type traits를 사용할 수 없는 조건이 있다

다음으로 nested requirement에 대해 조금 더 살펴보자

```
1 template<typename T>
2 concept C1 = sizeof(T) > 4;
3
4 template<typename T>
5 concept C2 = requires {
6     sizeof(T) > 4;
7 };
8
9 template<typename T>
10 concept C3 = requires {
11     requires sizeof(T) > 4;
12 };
13
14 int main() {
15     std::println("{} {}", C1<int>); // false
16     std::println("{} {}", C2<int>); // true
17     std::println("{} {}", C3<int>); // false
```

}

- C2는 T 타입에 대하여 `sizeof(T) > 4`를 사용할 수 있어야 한다는 의미이므로 `true`를 반환한다(원하는 행동이 아님)
- C3는 해당 조건식을 만족해야 한다는 의미이므로 `false`를 반환한다

nested requirement는 조금 더 복잡한 조건식을 `requires` 안에서 사용해야 할 때 유용하다.

```

1 template<typename T>
2 concept C1 = requires(T a) {
3     requires std::same_as<std::remove_reference_t<decltype(*a)>,
4         std::remove_pointer_t<T>>;
5     // 진짜 포인터를 사용하고 있는지 체크하는 코드
6 }
7
8 template<typename T>
9 concept C2 = requires {
10     std::same_as<std::remove_reference_t<decltype(*a)>,
11         std::remove_pointer_t<T>>;
12     // 위 표현식을 쓸 수 있는지 확인하는 코드
13 };
14
15 int main() {
16     std::println("{}", C1<int*>); // true
17     std::println("{}", C1<std::shared_ptr<int>>); // false
18
19     std::println("{}", C2<int*>); // true
20     std::println("{}", C2<std::shared_ptr<int>>); // true
21 }
```

6.5.3 concept example

간단한 컨셉 예제를 하나 만들어보자

```

1 template<typename T>
2 concept container = requires(T c) {
3     c.begin();
4     c.end();
5 }
6
7 int main() {
8     using T1 = std::vector<int>;
9     using T2 = int[3];
10
11     std::println("{}", container<T1>); // true
12     std::println("{}", container<T2>); // false
13 }
```

- `container<int[3]>`의 경우는 위 코드에 따르면 `false`를 반환한다
- 하지만 배열도 컨테이너로 보고 싶다면 어떻게 해야 할까?

```

1 template<typename T>
2 concept container = requires(T c) {
3     std::begin(c);
4     std::end(c);
5 }
6
7 int main() {
8     using T1 = std::vector<int>;
9     using T2 = int[3];
10
11     std::println("{}", container<T1>); // true
12     std::println("{}", container<T2>); // 여전히 false
13 }
```

-
- `std::begin()`, `std::end()`를 사용하면 배열도 컨테이너의 한 종류가 될 것 같다
 - 하지만 위 코드도 `container<int[3]>`는 `false`를 반환한다. 왜 그럴까?

```
1 void f1(int c[3]) {
2     std::println("{}", typeid(c).name());
3 }
4
5 void f2(int (c)[3]) {
6     std::println("{}", typeid(c).name());
7 }
8
9 int main() {
10     int x[3]={1,2,3};
11     f1(x); // int*
12     f2(x); // int[3]
13 }
```

- `f1(int c[3])` 도 타입을 확인해보면 `f1(int* c)`로 받는다
- 배열 그 자체로 받으려면 어떻게 해야 할까?
- `f2` 같이 받으면 배열 그 자체로 받는다

```
1 template<typename T>
2 concept container = requires(T& c) {
3     std::begin(c);
4     std::end(c);
5 };
6
7 int main() {
8     using T1 = std::vector<int>;
9     using T2 = int[3];
10
11     std::println("{}", container<T1>); // true
12     std::println("{}", container<T2>); // true
13 }
```

- `T`가 아닌 `T&`로 받아야 배열을 그대로 받을 수 있다

위 코드를 조금 더 발전시켜보자

```
1 template<typename T>
2 concept container = requires(T& c) {
3     std::begin(c);
4     std::end(c);
5 };
6
7 struct MyType {
8     void begin() {}
9     void end() {}
10 };
11
12 int main() {
13     using T = MyType;
14
15     std::println("{}", container<T>); // true
16 }
```

- `MyType`의 `begin()`, `end()`도 있지만 반환 타입이 반복자가 아닌 경우 이를 어떻게 구별해야 할까?
- `requires`는 `begin()`, `end()` 함수가 동작하는지만 판단하므로 `true`가 나온다

반환 타입이 반복자인 조건을 추가한다

```
1 template<typename T>
2 concept container = requires(T& c) {
```

```

3     { std::begin(c) }-> std::input_or_output_iterator;
4     { std::end(c) }-> std::input_or_output_iterator;
5 };

```

- simple에서 compound require로 변경해야 한다

조금 더 간단한 버전도 존재한다

```

1 template<typename T>
2 concept container = requires(T& c) {
3     std::ranges::begin(c);
4     std::ranges::end(c);
5 };

```

- `std::begin(c)` (c++11) : `c.begin()`의 반환 타입을 조사하지 않는다
- `std::ranges::begin(c)` (c++20) : `c.begin()`의 반환 타입이 반복자 조건을 만족하지 않으면 컴파일 에러가 발생한다
- 위 사실을 토대로 `std::ranges::begin(c)`을 사용해도 된다

c++ 표준이 제공하는 미리 만들어진 container 컨셉은 없을까?

```

1 template<typename T>
2 concept container = requires(T& c) {
3     std::ranges::begin(c);
4     std::ranges::end(c);
5 };
6
7 int main() {
8     std::println("{}", container<int[3]>);
9     std::println("{}", std::ranges::range<int[3]>);
10 }

```

- c++ 표준의 `std::ranges::range`라는 컨셉은 구현이 우리가 직접 구현한 `container`와 동일하다
- 따라서 컨테이너 여부를 판단할 때 사용할 수 있다
- 왜 이름이 `continaer`가 아닌 `range`일까?
- **Container**(c++98~) : `vector, list, deque, ...` 자원(저장하는 데이터)를 소유
- **Views**(c++20~) : `take_veiw, reverse_view, ...` 자원을 소유하지 않고 컨테이너에 대한 view 역할
- **Range = Container + View**
- Range : 범위(begin, end)를 구할 수 있는 객체

6.5.4 requires clauses

requires clauses라는 개념에 대해 살펴보자

```

1 struct Point{
2     int x,y;
3 };
4
5 template<typename T>
6 T add(const T& a, const T& b){
7     return a+b;
8 }
9
10 int main() {
11     Point pt1{1,1};
12     Point pt2{1,1};
13     auto pt3 = add(pt1, pt2); // error
14 }

```

- `Point`에는 덧셈 연산자가 오버로딩 되어 있지 않으므로 에러가 발생한다
- 아예 처음부터 덧셈이 지원되지 않는 타입은 `add` 템플릿 자체를 사용하지 못하게 할 수 없을까?
- 이런 상황에서 requires clauses 문법이 사용된다

```

1 struct Point{
2     int x,y;
3 };
4
5 template<typename T>
6 concept addable = requires(T*& a, T& b) {
7     a+b;
8 };
9
10 template<typename T> requires addable<T>
11 T add(const T& a, const T& b){
12     return a+b;
13 }
14
15 int main() {
16     Point pt1{1,1};
17     Point pt2{1,1};
18     auto pt3 = add(pt1, pt2); // error 아래 컴파일이 되지 않는다
19 }
```

- requires clauses가 없는 경우 add 인스턴스가 생성되고 실제 연산 수행 시 에러 발생
- requires clauses가 있는 경우 add 인스턴스가 생성되지 않고 컴파일 에러 발생
- 둘 다 모두 에러가 발생하는데 requires clauses를 꼭 사용해야 할까?

requires clauses의 장점

1. 더 좋은 에러 메세지
2. 의도치 않은 버그를 막을 수 있다
3. 조건에 따른 오버로딩
4. 조건을 만족하는 경우 멤버 함수 제공

또 다른 예제를 살펴보자

```

1 template<typename T> requires true // bool literal
2 void f1(T a) {}
3
4 template<typename T> requires std::is_pointer_v<T> // type traits
5 void f2(T a) {}
6
7 template<typename T> requires std::convertible_to<T, bool> // standard concept
8 void f3(T a) {}
9
10 template<typename T> requires (sizeof(T) > 4)
11 void f4(T a) {}
12
13 template<typename T> requires requires(T a, T b) { {a+b} -> std::same_as<T>; }
14 void f5(T a) {}
```

- requires 키워드의 2가지 용도
- (1) requires expression(표현식)
- (2) requires clauses(절)
- 하나의 키워드가 두 개의 다른 용도로 쓰이고 있음에 유의한다(서로 다른 기능임)

requires clauses의 위치는 다음과 같이 두 곳에 위치할 수 있다

```

1 template<typename T> requires true
2 void f1(T a) {}
3
4 template<typename T>
5 void f2(T a) requires true
6 {}
```

requires clauses를 사용했을 때 장점에 대해 자세히 살펴보자 (더 좋은 에러 메세지)

```

1 class Animal {};
2 class Dog : public Animal {};
3
4 template<typename T>
5 class smart_pointer {
6     T* obj;
7     public:
8         explicit smart_pointer(T* p = nullptr) : obj(p) {}
9
10    // generic 생성자
11    template<typename U>
12        smart_pointer(const smart_pointer<U>& other) : obj(other.obj) {}
13    template<typename> friend class smart_pointer;
14 }
15
16 int main() {
17     smart_pointer<Dog> sp1{ new Dog };
18     smart_pointer<Animal> sp2 = sp1; // generic 생성자 덕분에 사용 가능
19
20     smart_pointer<int> sp3 = sp1; // 만약 이런 상황이라면?
21 }
```

- sp3 = sp1을 가리키면 obj(other.obj)에서 컴파일 에러가 발생한다
- 클래스를 사용하는 사용자 입장에서는 해당 에러가 어디서 발생했는지 바로 파악하기 힘들다
- requires clauses를 활용하여 조금 더 나은 에러 메세지를 만들 수 있다

```

1 template<typename T>
2 class smart_pointer {
3     T* obj;
4     public:
5         explicit smart_pointer(T* p = nullptr) : obj(p) {}
6
7     // generic 생성자
8     template<typename U>
9         requires std::convertible_to<U*, T*>
10        smart_pointer(const smart_pointer<U>& other) : obj(other.obj) {}
11    template<typename> friend class smart_pointer;
12 }
```

- 에러 메세지가 사용자 코드 sp3 = sp1에서 발생하므로 상대적으로 쉽게 에러 원인을 알 수 있다

다음으로 두번째 장점을 살펴보자 (안전한 코드 작성 가능)

```

1 class Person {
2     std::string name;
3     public:
4         template<typename T>
5         void set_name(T&& n) {
6             name = std::forward<T>(n);
7         }
8     };
9
10 int main() {
11     Person p;
12     p.set_name(10); // 이름을 줘야하는데 10을 줬다. 그런데 에러가 발생하지 않는다
13 }
```

- set_name을 만들 때 forward reference(T&&)를 사용하면 move를 지원하는 setter를 간단하게 만들 수 있다
- int는 string으로 변환될 수 있기 때문에 set_name(10)이 에러가 발생하지 않는다

```

1 class Person {
2     std::string name;
3     public:
```

```

4     template<typename T>
5         requires std::convertible_to<T, std::string>
6     void set_name(T&& n) {
7         name = std::forward<T>(n);
8     }
9 };

```

- 위와 같이 생각지도 못한 버그를 사전에 방지할 수 있다

또 다른 세번째 장점을 살펴보자 (조건을 만족하는 경우만 멤버함수 제공)

```

1 template<typename T>
2 class Object{
3     public:
4         void f1() {}
5         void f2() requires std::integral<T> {} // 함수 뒤에도 requires clauses를 사용할 수 있다
6     };
7
8     int main() {
9         Object<int> o1;
10        Object<double> o2;
11
12        o1.f1();
13        o1.f2();
14        o2.f1();
15        o2.f2(); // error
16    }

```

- Object 클래스는 requires가 없지만 f2 함수에는 requires가 존재할 수 있다
- 이와 같이 조건을 만족하는 경우에만 멤버 함수를 제공할 수 있다

requires clauses를 활용하여 클래스 상속을 제어하는 예제 코드를 보자

```

1 template<typename T>
2 concept F2 = requires {
3     typename T::i_want_f2;
4 };
5
6 template<typename T>
7 class Base {
8     public:
9         void f1() {}
10        void f2() requires F2<T> {};
11    };
12
13 class D1 : public Base<D1> {
14 };
15
16 class D2 : public Base<D2> {
17     public:
18     using i_want_f2 = void;
19 };
20
21 int main() {
22     D1 d1;
23     D2 d2;
24
25     d1.f1();
26     d1.f2(); // error!
27     d2.f1();
28     d2.f2();
29 }

```

- D1에는 i_want_f2가 없기 때문에 f2가 상속되지 않는다
- 반면에 D2에는 i_want_f2가 존재하므로 모든 멤버 함수를 상속받는다

- 기반 클래스 Base와 Derived 이름을 넘겨주는 기법은 CRTP라고 불린다. 다른 섹션에서 자세히 설명한다

마지막 네번째 requires clauses의 장점에 대해 살펴보자(조건에 따른 함수 템플릿 선택)

```
1 template<typename T> requires (std::is_pointer_v<T>)
2     void printv(const T& a) {
3         std::cout << "pointer" << std::endl;
4     }
5
6 template<typename T> requires (!std::is_pointer_v<T>)
7     void printv(const T& a) {
8         std::cout << "not pointer" << std::endl;
9     }
10
11 int main() {
12     int n = 0;
13     printv(n);
14     printv(&n);
15 }
```

- requires 키워드를 사용하여 포인터 조건에 따라 다른 함수를 실행할 수 있다

또 다른 네번째 장점을 알 수 있는 중요한 예제를 살펴보자

```
1 template<typename T> concept C1 = std::convertible_to<T, bool>
2 template<typename T> concept C2 = sizeof<T> > 2;
3 template<typename T> concept C3 = std::integral<T>;
4
5 template<typename T> concept C4 = C1<T> && C2<T>
6 template<typename T> concept C5 = C1<T> && C2<T> && C3<T>
7
8 template<typename T> requires C1<T>
9     void f1(T a) { std::println("f1-C1"); }
10
11 template<typename T> requires C4<T>
12     void f1(T a) { std::println("f1-C4"); }
13
14 template<typename T> requires C5<T>
15     void f1(T a) { std::println("f1-C5"); }
16
17 int main() {
18     f1(10);    // 3개 다 만족한다. 보다 많은 조건을 만족하는 함수가 호출되므로 f1-C5가 출력된다
19     f1(3.4);  // 2개 만족, f1-C4
20     f1('A');   // 1개 만족, f1-C1
21 }
```

6.6 Variadic template

6.6.1 variadic template

c++11부터 추가된 가변 인자 템플릿(variadic template)에 대해서 살펴보자. 가변 인자 템플릿과 관련되어 알아야 할 내용들은 아래와 같다

- 가변 인자 클래스 템플릿
- 가변 인자 함수 템플릿
- parameter pack
- pack expansion
- fold expression (c++17)

가변 인자 클래스 템플릿의 기본 모양에 대해 알아보자

```

1 template<typename T1, typename T2> class pair {
2 };
3
4 template<typename ... Ts> class tuple {
5 };
6
7 int main() {
8     pair<int, double> p2;
9
10    tuple t0;           // ok (c++17)
11    tuple<> t1;         // ok
12    tuple<int> t2;       // ok
13    tuple<int, double> t3; // ok (Ts : int, pack)
14 }
```

- 가변 인자 템플릿 : 템플릿 파라미터에 ...을 사용하는 기술
- 템플릿 사용 시 타입 인자의 갯수에 제한이 없다
- T1, T2 : 템플릿 파라미터
- Ts : 템플릿 파라미터 팩(template parameter pack)

다음으로 가변 인자 함수 템플릿의 기본 모양에 대해 알아보자

```

1 template<typename T1, typename T2> void f1(T1 args1, T2 arg2) {
2 };
3
4 template<typename ... Ts> void f2(Ts ... args) {
5 };
6
7 int main() {
8     f1<int>(3);           // error
9     f1<int, double>(3, 3.4); // ok
10    f1(3, 3.4);          // ok 함수 인자보고 타입 추론
11
12    f2<int>(3);
13    f2<int, double, char>(3, 3.4, 'A');
14    f2(3);                // ok
15    f2(3, 3.4, 'A');      // ok 타입 인자 생략 가능
16 }
```

파일 - `const`를 사용하고 싶은 경우 `void f2(const T2& ... args)`와 같이 사용하면 된다

```

1 void f3( ... ) {}
2
3 int main() {
4     f3(3);
5     f3(3, 3.4, 'A');
6 }
```

- f3는 가변 인자 템플릿이 아닌 가변 인자 함수이다
- 가변 인자 함수 템플릿과 달리 `f3(3)`, `f3(3, 3.4, 'A')` 모두 같은 함수가 호출된다
- f3는 인자 타입에 대한 정보가 없으므로 이를 알기 위해서는 복잡한 방법이 필요한 반면, 가변 인자 템플릿은 바로 확인할 수 있다

6.6.2 Pack expansion

파라미터 팩이라는 개념에 대해 살펴보자

```

1 template<typename ... Ts> void fn(const Ts& ... args) {
2     std::cout << sizeof...(Ts) << std::endl;
3     std::cout << sizeof...(args) << std::endl;
4 };
5 
```

```

6 int main() {
7     fn(1, 3.4, 'A'); // fn(int, double, char);
8     fn();           // fn();
9 }
```

- Ts : int, double, char : 템플릿 파라미터 팩
- args: 1, 3.4, 'A' : 함수 파라미터 팩
- sizeof... : 파라미터 팩 안에 있는 요소의 개수를 구하는 연산자

pack expansion이라는 개념에 대해 살펴보자

```

1 void foo(int a, int b, int c) {
2     std::cout << a << ", " << b << ", " << c << std::endl;
3 }
4
5 template<typename ... Ts>
6 void fn(Ts ... args) {
7     // args: 1, 2, -3
8     // foo( args ); // error
9
10    foo( args... );
11
12    foo( ++args... ); // ++e1, ++e2, ++e3 각 요소를 증가한 후 넣어달라
13        // foo(2, 3, -2)
14
15    foo( abs(args)... ); // foo(1, 2, 3)
16 }
17
18 int main() {
19     fn(1, 2, -3);
20 }
```

- foo(args) : args가 바로 함수의 파라미터로 들어가지는 못한다
- foo(args...) : args 안에 있는 변수들을 풀어서 넣어달라는 의미
- pack expansion : 파라미터 팩 안에 있는 모든 요소를 콤마(,)를 사용하새 순서대로 나열하는 방법
- 팩 뿐만 아니라 팩 이름을 사용하는 패턴에도 사용 가능하다
- 팩 이름을 사용하는 패턴 (++args..., abs(args)...)

pack expansion 관련하여 다른 예제를 살펴보자

```

1 template<typename T> void print(const T& value) {
2     std::cout << value << std::endl;
3 }
4
5 template<typename ... Ts> void fn(Ts ... args) {
6     print( args... ); // print(3, 3.4, 'A') - error 발생
7
8     print( args )...; // print(3), print(3.4), print('A')를 기대했으나 error 발생
9
10    int dummy[] = { print( args )... }; // error 중괄호 안에 있으므로 동작하긴 하지만 print가 void를
11        반환하므로 에러가 발생한다
12
13    int dummy2[] = { (print(args), 0)... }; // ok (a, b) a 문장을 실행하고 b 값을 사용하겠다는 의미
14
15 int main() {
16     fn(3, 3.4, 'A');
17 }
```

- pack expansion loci : 모든 문맥에서 pack expansion이 가능한 것은 아님
- 함수 인자를 전달하는 괄호 안 또는 초기화되는 문맥에서 사용 가능하다

dummy2까지 하면 모든 경우에 대해 정상적으로 실행될까?

```

1 template<typename ... Ts> void fn(Ts ... args) {
```

```

2     int dummy2[] = { (print(args), 0)... }; // error 아무 값도 입력받지 못하면 에러 발생
3
4     int dummy3[] = {0 , (print(args), 0)... }; // ok
5
6
7     int main() {
8         fn();
9     }

```

- fn() 인자가 하나도 없는 경우까지 처리해주면 dummy3 같이 된다
- `int dummy3[] = {0 , (print(args), 0)... };`; C++ 초기기에 많이 사용했던 가변 인자를 출력하는 코드였다. 현재는 더 나은 구현들이 사용된다

다음으로 pack expansion이 가능한 다양한 경우들에 대해 알아보자

```

1     template<typename ... Ts> void fn(Ts ... args) {
2         // Ts : int, int
3         // args: 1, 2
4
5         // #1 중괄호 초기화
6         int arr[] = { args... };
7
8         // #2 함수 호출 시
9         int ret = std::min( args... );
10
11        // #3 객체를 만들 때 함수 인자로 사용
12        // #4 템플릿 타입 인자로 사용
13        std::pair<Ts...> p1( args... );
14
15        // #5 람다 캡쳐식에 사용
16        auto f1 = [ args... ] { return std::min(args...); };
17    }
18
19    int main() {
20        fn(1,2);
21    }

```

- pack expansion을 사용할 수 있는 경우
- (1) 중괄호 초기화
- (2) 함수 호출 시
- (3) 객체를 만들 때 함수 인자로
- (4) 템플릿 타입 인자로 사용
- (5) 램다 표현식에 사용

```

1     template<typename ... Ts> struct Outer {
2         // #6 템플릿 파라미터 리스트로 사용
3         template<Ts ... value>
4             struct Inner {};
5     };
6
7     Outer<int, double>::Inner<3, 3.4> in;
8
9     // -----
10    class A{};
11    class B{};
12
13    // #7 기반 클래스에 멤버 이니셜라이저를 실행할 때
14    template<typename... Ts>
15    class X : public Ts... { // X가 여러 클래스로부터 상속 받음
16        public:
17            X(const Ts&... args) : Ts(args)... {}
18    };

```

```

21 int main() {
22     A a;
23     B b;
24     X<A,B> x(a,b);
25 }
```

- (6) 템플릿 파라미터 리스트로 사용
- (7) 기반 클래스에 멤버 이니셜라이저를 실행할 때
- 6번은 가변 인자 템플릿 안에 또 다른 템플릿 파라미터를 만드는 경우 사용한다

6.6.3 variadic template recursion

가변 인자 템플릿에서 재귀를 사용하는 경우에 대해 살펴보자

```

1 template<typename T, typename ... Ts>
2 void print(T value, Ts ... args) {
3     std::print("{}", value);
4
5     if constexpr ( sizeof...(args) > 0 )
6         print(args...); // print(2,3); recursion 사용
7             // print(3);
8             // print();
9 }
10
11 int main() {
12     print(1, 2, 3); // value : 1
13             // args : 2, 3
14 }
```

- pack expansion을 사용하지 않고 각각의 인자에 접근할 수 있는 방법은 없을까?
- 모든 인자를 variadic template으로 받지 말고 **첫번째 인자는 독립된 인자**로 받아야 한다.
- 그 다음 재귀적으로 호출한다. (재귀의 종료가 필요함)
- c++14 이전에는 인자가 없는 print() 함수를 제공한다
- c++17 이후에는 재귀의 종료를 위해 `if constexpr(sizeof...(args)>0)`을 사용할 수 있다
- 하지만 위 함수가 정말 재귀적으로 함수를 호출하는 것일까? 아니다. **인자의 갯수가 다른 여러 print 함수를 호출**하는 것이다

재귀를 사용한 간단한 예제를 보자

```

1 int sum() { return 0; } // base case (c++14)
2
3 template<typename T, typename ... Ts>
4 auto sum(T value, Ts ... args) {
5     return value + sum(args...);
6 }
7
8 int main() {
9     auto s = sum(1,2,3,4,5);
10    std::println("{}", s);
11 }
```

- sum을 만드는 방법이 위 방법만 존재하는게 아니라 fold expression이라는 방법을 통해서도 만들 수 있다

6.6.4 fold expression

이번에는 fold expression이라는 문법에 대해 살펴보자

```

1 template<typename ... Ts>
2 auto sum(Ts ... args) {
3     auto s = ( ... + args ); // fold expression
4     return s;
5 }
6
```

```
7 int main() {
8     auto ret = sum(1,2,3,4,5);
9     std::println("{}", s);
10 }
```

- fold expression(c++17) : 파라미터 팩 안에 있는 모든 요소들에 대해 이항 연산을 하고 싶은 경우 사용한다
- 내부적으로 팔호를 사용하여 연산된다
- `auto ret = sum()`에 대해서 현재 코드는 컴파일 에러가 발생하는데 만약 이를 방지하고 싶다면 `auto s = (0 + ... + args)`와 같이 입력하면 된다

fold expression은 4가지 형태로 사용 가능하다

```
1 template<typename ... Ts>
2     auto sum(Ts ... args) {
3         auto s1 = ( args + ... );      // (E1+(E2+(E3+(E4+E5))))
4         auto s2 = ( ... + args );    // (((E1+E2)+E3)+E4)+E5
5         auto s3 = ( args + ... + 0 ); // (E1+(E2+(E3+(E4+(E5+init))))) )
6         auto s4 = ( 0 + ... + args ); // ((((init+E1)+E2)+E3)+E4)+E5
7
8     return s1;
9 }
10
11 int main() {
12     auto ret = sum(1,2,3,4,5);
13     std::println("{}", s);
14 }
```

- (`pack op ...`) : unary right fold
- (`... op pack`) : unary left fold
- (`pack op ... op init`) : binary right fold
- (`init op ... op pack`) : binary left fold
- 팔호가 끊이는 순서가 다르기 때문에 형태를 4가지로 구분한다 - 반드시 ()를 붙여야 한다

또 다른 예제를 살펴보자

```
1 template<typename ... Ts>
2     void show(Ts ... args) {
3         ( std::cout << ... << args );
4     }
5
6 int main() {
7     show(1, 2, 3);
8 }
```

- () 연산이 있으므로 fold expression이다
- `std::cout << ... << args`이므로 init op ... op pack이 되어 binary left folder 형태가 된다
- 팩의 앞쪽 요소부터 << 연산을 적용한다
- (((`std::cout << 1`)<< 2)<< 3)

팩의 이름을 사용하는 다양한 패턴에도 적용 가능하다

```
1 template<typename ... Ts>
2     void show(Ts ... args) {
3         ( std::print("{} ", args) , ... );
4     }
5
6 int main() {
7     show(1, 2, 3);
8 }
```

- (`std::print("{} ", args)`, ...) 표현에서 , 가 이항 연산자이다
- (`pack op ...`) 형태인 unary right fold이다
- (`std::print("{} ", 1), (std::print("{} ", 2), std::print("{} ", 3)))`)

```

1 template<typename T, typename ... Ts>
2 void push_vec(std::vector<T>& v, Ts&& .. args) {
3     ( v.push_back(std::forward<Ts>(args)), ... );
4 }
5
6 int main() {
7     std::vector<int> v;
8
9     push_vec(v, 1,2,3,4,5);
10
11    for(auto e : v) {
12        std::print("{} ", e);
13    }
14 }
```

- 가변 인자 템플릿과 fold expression을 사용하여 벡터에 모든 값을 넣을 수 있다

6.6.5 variadic template example

가변 인자 템플릿을 사용하는 여러 예제들에 대해 살펴보자

```

1 class stopwatch { ... };
2
3 void f1(int n) { std::this_thread::sleep_for(2s); }
4 void f2(double d) { std::this_thread::sleep_for(3s); }
5
6 int main() {
7     {
8         stopwatch sw;
9         f1(5);
10    }
11    {
12        stopwatch sw;
13        f2(3.4);
14    }
15 }
```

- 임의 함수의 수행시간을 측정하기 위해 stopwatch 클래스를 작성하였다. stopwatch는 스코프를 벗어나면 파괴될 때 측정한 시간을 출력한다

stopwatch 기능을 자주 사용하게 되어자 이를 스코프 방식으로 작성하기보다 하나의 깔끔한 함수로 만들어 보자

```

1 void f1(int n) { std::this_thread::sleep_for(2s); }
2 void f2(double d) { std::this_thread::sleep_for(3s); }
3
4 template<typename F, typename T>
5 void chronometry(F f, T arg) {
6     stopwatch sw;
7     f(arg);
8 }
9
10 int main() {
11     chronometry(f1, 5);
12     chronometry(f2, 3.4);
13 }
```

- chronometry는 현재 인자가 1개인 함수 밖에 받지 못한다
- 여러 인자를 가진 함수에 대해서도 시간을 측정하고 싶은 경우 어떻게 해야 할까?

```

1 void f1(int a, int b) { std::this_thread::sleep_for(2s); }
2 void f2(double d) { std::this_thread::sleep_for(3s); }
3
```

```

4 template<typename F, typename ... Ts>
5 void chronometry(F f, Ts ... args) {
6     stopwatch sw;
7     f(args...);
8 }
9
10 int main() {
11     chronometry(f1, 5, 6);
12     chronometry(f2, 3.4);
13 }
```

실제 perfect forwarding 등을 고려하여 제대로 만드려면 다음과 같이 복잡해진다.

```

1 template<typename F, typename ... Ts>
2 decltype(auto) void chronometry(F&& f, Ts&& ... args) {
3     stopwatch sw;
4     return std::invoke( std::forward<F>(f), std::forward<Ts>(args)... );
5 }
```

6.6.6 make tuple

가변 인자 템플릿을 활용하여 c++ 표준에 있는 tuple과 유사한 클래스를 만들어보자

```

1 int main() {
2     std::vector<int> v{1,2,3}; // 전부 같은 타입
3
4     std::tuple<int, double, char> t(3, 3.4, 'A'); // 서로 다른 타입
5
6     std::get<1>(t) = 9.9;
7
8     std::cout << std::get<0>(t) << std::endl; // 3
9     std::cout << std::get<1>(t) << std::endl; // 9.9
10 }
```

- `std::tuple` (c++11): 임의 갯수의 서로 다른 타입의 객체를 저장할 수 있는 타입

`std::tuple` 객체를 최대한 비슷하게 만들어보자

```

1 // primary template
2 template<typename ... Ts> struct tuple {
3     static constexpr int N = 0;
4 };
5
6 // partial specialization (첫번째 인자가 있는 경우)
7 template<typename T, typename ... Ts> struct tuple<T, Ts...> {
8     T value;
9     tuple() = default;
10    tuple(const T& v) : value(v) {}
11    static constexpr int N = 1;
12 };
13
14 int main() {
15     tuple<> t0;
16     tuple<char> t1;
17     tuple<double, char> t2;
18     tuple<int, double, char> t3;
19 }
```

- 가변 인자 템플릿 사용
- 저장하는 요소의 개수를 관리하는 `static` 멤버 `N` 제공
- 템플릿 인자가 하나 이상 있는 경우를 위해 부분 특수화 진행

현재 위 코드는 템플릿 인자가 1개인 경우만 사용 가능하다. 이를 `N`개로 확장하려면 어떻게 해야 할까?

```

1 template<typename T, typename ... Ts>
2 struct tuple<T, Ts...> : public tuple<Ts...>
3 {
4     T value;
5     tuple() = default;
6     tuple(const T& v) : value(v) {}
7     static constexpr int N = tuple<Ts...>::N + 1; // 개수 관리
8 };

```

- 재귀 상속을 이용해서 여러 값을 받을 수 있다

가독성을 위한 코드와 생산자 코드를 조금 더 다듬으면 다음과 같이 된다

```

1 template<typename T, typename ... Ts>
2 struct tuple<T, Ts...> : public tuple<Ts...>
3 {
4     using base = tuple<Ts...>;
5
6     T value;
7     tuple() = default;
8
9     // 여러 인자를 받도록 생성자 변경
10    template<typename ... Types> // (디폴트 생성자까지 처리하도록)
11        tuple(const T& v, const Types ... args) : value(v), base(args...) {}
12
13    static constexpr int N = base::N + 1;
14 };
15
16 int main() {
17     tuple<int, double, char> t3(3, 3.4, 'A');
18     tuple<int, double, char> t4(3, 3.4); // 디폴트 파라미터
19 }

```

`std::move`까지 처리하도록 forward reference를 적용하면 생성자는 다음과 같다

```

1 template<typename A, typename ... Types>
2 tuple(A&& v, Types&& ... args) : value(std::forward<A>(v)), base(std::forward(args)...){}

```

지금까지 모든 걸 고려하여 템플릿 클래스를 만들어보자

```

1 // primary template
2 template<typename ... Ts> struct tuple {
3     static constexpr int N = 0;
4 };
5
6 // partial specialization
7 template<typename T, typename ... Ts>
8 struct tuple<T, Ts...> : public tuple<Ts...>
9 {
10     using base = tuple<Ts...>;
11
12     T value;
13     tuple() = default;
14
15     template<typename A, typename ... Types>
16         tuple(A&& v, Types&& ... args) : value(std::forward<A>(v)), base(std::forward(args)...){}
17
18     static constexpr int N = base::N + 1;
19 };
20
21 int main() {
22     tuple<int, double, char> t3(3, 3.4, 'A');
23     tuple<int, double, char> t4(3, 3.4); // 디폴트 파라미터
24 }

```

6.6.7 make get

앞서 구현한 tuple의 각 요소에 접근하는 방법에 대해 알아보자

```
1 int main() {
2     std::tuple<int, double, char> t(3, 3.4, 'A');
3
4     std::cout << "[]" << std::get<0>(t); // 3
5     std::cout << "[]" << std::get<1>(t); // 3.4
6     std::cout << "[]" << std::get<2>(t); // 'A'
7
8     std::get<0>(t) = 20;
9
10    std::cout << "[]" << std::get<0>(t); // 20
11 }
```

- `std::get` : 튜플의 각 요소를 꺼내기 위해 사용

`std::get`을 구현하기 전에 알아야 할 개념이 있다

```
1 struct Base {
2     double value = 3.4;
3 };
4
5 struct Derived : Base {
6     int value = 3;
7 };
8
9 int main() {
10     Derived d;
11     std::cout << "[]" << d.value; // 3;
12     std::cout << "[]" << static_cast<Base&>(d).value; // 3.4
13 }
```

- 동일한 이름의 변수가 기반 클래스에도 있고 상속 클래스에도 있는 경우 `static_cast`를 사용하여 기반 클래스 변수에 접근할 수 있다

- `static_cast<Base>`를 하면 임시객체가 만들어지므로 반드시 `static_cast<Base&>`로 해야 한다

앞서 구현한 튜플은 상속을 사용하여 여러 타입들을 보관하고 있었다

```
// tuple 코드 생략 ...
1
2
3 int main() {
4     tuple<int, double, char> t(3, 3.4, 'A'); // t는 현재 value에 3만 담고 있다
5
6     std::cout << "[]" << t.value; // 3
7     std::cout << "[]" << static_cast<tuple<char>&>(t).value; // 'A'
8 }
```

- 위 코드를 바탕으로 `get`을 어떻게 구현해야 할지 생각해보자

- `get`은 추가적으로 `auto n = get<0>(t);, get<0>(t)= 10;` 같은 코드들이 동작해야 한다

```
1 template<std::size_t N, typename TP>
2     TP의 N번째 요소의 타입&
3     get(TP& t) {
4         return static_cast<TP의 N번째 Base 타입&>(t).value;
5     }
```

- 결국 `get`을 구현하려면 임의의 튜플 타입 `TP`에 대해서 **N번째 요소의 타입**과 **N번째 기반 클래스의 타입**을 알아야 한다

`get`을 직접 구현해보자

```
1 template<typename TP>
```

```

2 void test(TP& tp) {
3     // TP : tuple<int, double, char>
4
5     typename tuple_element<0, TP>::type n;
6
7     std::println("{}", typeid(n).name());
8 }
9
10 int main() {
11     tuple<int, double, char> t(3, 3.4, 'A');
12     test(t);
13 }
```

- 3 : `int` 타입이며 0번째 요소이다. 0번째 기반 클래스가 보관하고 있다
- 3.4 : `double` 타입이며 1번째 요소이다. 1번째 기반 클래스가 보관하고 있다
- 'A' : `char` 타입이며 2번째 요소이다. 2번째 기반 클래스가 보관하고 있다
- `tuple_element<N, TP>` : 투플 TP의 N번째 요소의 타입을 구하는 템플릿(c++ 표준에도 동일한 이름으로 제공됨)

(1) primary template을 만들자

```

1 template<std::size_t N, typename TP>
2 struct tuple_element {
3     using type = ?; // 현재는 구할 수 없다
4 };
5
6 template<typename TP>
7 void test(TP& tp) {
8     // TP : tuple<int, double, char>
9     typename tuple_element<0, TP>::type n;
10
11     std::println("{}", typeid(n).name());
12 }
13
14 int main() {
15     tuple<int, double, char> t(3, 3.4, 'A');
16     test(t);
17 }
```

(2) TP의 N번째 타입을 구할 수 있도록 부분특수화해야 한다

```

1 // primary template
2 template<std::size_t N, typename TP>
3 struct tuple_element;
4
5 // partial specialization
6 template<typename T, typename ... Ts>
7 struct tuple_element<0, tuple<T, Ts...>>
8 {
9     using type = T; // 0번째 변수의 타입
10 }
```

- 먼저 N이 아닌 0번째를 구할 수 있도록 부분특수화를 해야 한다
- primary 버전은 최종적으로 사용하지 않게 되므로 선언만 있어도 된다

(3) N != 0인 경우의 타입을 구할 수 있도록 부분 특수화 해야한다

```

1 // primary template
2 template<std::size_t N, typename TP>
3 struct tuple_element;
4
5 // partial specialization (N==0)
6 template<typename T, typename ... Ts>
7 struct tuple_element<0, tuple<T, Ts...>>
8 {
9     using type = T;
```

```

10     };
11
12 // partial specialization (N!=0)
13 template<std::size_t N, typename T, typename ... Ts>
14 struct tuple_element<N, tuple<T, Ts...>>
15 {
16     using type = tuple_element<N-1, tuple<T, Ts...> >::type; // N번째 변수의 타입
17 };

```

- tuple<T, Ts...>의 N번째 타입은 tuple<Ts...>의 N-1번째 타입이다(T가 제거된 것이 핵심)

(4) 0번째 요소의 타입이 아닌 0번째 요소를 저장하는 tuple type을 구해야 한다

```

1 // primary template
2 template<std::size_t N, typename TP>
3 struct tuple_element;
4
5 // partial specialization (N==0)
6 template<typename T, typename ... Ts>
7 struct tuple_element<0, tuple<T, Ts...>>
8 {
9     using type = T;
10    using tuple_type = tuple<T, Ts...>;
11 };
12
13 // partial specialization (N!=0)
14 template<std::size_t N, typename T, typename ... Ts>
15 struct tuple_element<N, tuple<T, Ts...>>
16 {
17     using type = tuple_element<N-1, tuple<T, Ts...> >::type;
18     using tuple_type = tuple_element<N-1, tuple<T, Ts...> >::tuple_type;
19 };
20
21 template<typename TP>
22 void test(TP& tp) {
23     typename tuple_element<2, TP>::type n;
24     std::println("{}", typeid(n).name());
25
26     typename tuple_element<2, TP>::tuple_type tn;
27     std::println("{}", typeid(tn).name());
28 }
29
30 int main() {
31     tuple<int, double, char> t(3, 3.4, 'A');
32     test(t);
33 }

```

- tuple<T, Ts...> : 0번째 요소의 타입은 T, 0번째 요소를 저장하는 tuple type은 자기 자신이다(tuple<T, Ts...>)

지금까지 배운 내용을 모두 결합해서 get을 구현해보자

```

1 // primary template
2 template<std::size_t N, typename TP>
3 struct tuple_element;
4
5 // partial specialization (N==0)
6 template<typename T, typename ... Ts>
7 struct tuple_element<0, tuple<T, Ts...>>
8 {
9     using type = T;
10    using tuple_type = tuple<T, Ts...>;
11 };
12
13 // partial specialization (N!=0)
14 template<std::size_t N, typename T, typename ... Ts>
15 struct tuple_element<N, tuple<T, Ts...>>

```

```

16   {
17     using type = tuple_element<N-1, tuple<T, Ts...> >::type;
18     using tuple_type = tuple_element<N-1, tuple<T, Ts...> >::tuple_type;
19   };
20
21 template<std::size_t N, typename TP>
22 typename tuple_element<N, TP>::type&
23 get(TP& t) {
24   return static_cast<typename tuple_element<N,TP>::tuple_type&>(t).value;
25 }
26
27 int main() {
28   tuple<int, double, char> t(3, 3.4, 'A');
29
30   get<0>(t) = 10;
31
32   std::println("{0}", get<0>(t)); // 10
33   std::println("{0}", get<1>(t)); // 3.4
34   std::println("{0}", get<2>(t)); // 'A'
35 }
```

- `get` 자체의 구현은 어렵지 않지만 요소의 타입과 기반 클래스의 타입을 구하는 과정이 복잡한 편이다
- C++ 표준의 `std::tuple_element`는 `type` 멤버만 제공하고 `tuple_type`은 제공하지 않는다
- `std::tuple`, `std::get`의 구현은 g++, cl, clang++, xcode 등 다양한 환경에서 자신들만의 방법으로 구현되어 있다

6.7 CRTP

6.7.1 CRTP

C++에서 유명한 기술 중 하나인 CRTP(Curiously Recurring Template Pattern)라는 기술에 대해 살펴보자

```

1  template<typename T>
2  class Base{
3    public:
4      void fn() {
5        // 여기서 파생 클래스 이름(Derived)를 사용할 수는 없을까?
6        T obj;
7        std::cout << typeid(T).name() << std::endl; // Derived
8      }
9  };
10
11 class Derived : public Base<Derived>
12 {
13 };
14
15
16 int main() {
17   Derived d;
18   d.fn();
19 }
```

- **CRTPO(Curiously Recurring Template Pattern)**: 기반 클래스에서 파생 클래스의 클래스 이름을 사용할 수 있게 하는 기술(뜻을 풀어보면 신기하게도 되풀이되는 템플릿 패턴이라는 뜻이다)
- CRTP의 핵심 : 기반 클래스를 템플릿으로 만들고 파생 클래스를 만들 때 **자신의 클래스 이름을 템플릿 인자로 전달**한다

CRTP를 사용하는 예제에 대해 살펴보자

```

1  class Window{
2    public:
3      void event_loop() {
4        Click();
5      }
6      virtual void Click() { }
```

```

7   };
8
9   class MainWindow : public Window {
10  public:
11    void Click() override { }
12  };
13
14 int main() {
15   MainWindow w;
16   w.event_loop();
17 }
```

- `w.event_loop()`를 호출하면 `MainWindow`의 `Click()` 함수가 호출된다
- GUI 이벤트를 가상함수로 처리하는 경우 GUI 이벤트 자체가 매우 많기 때문에 [가상함수 테이블의 크기에 대한 메모리 오버헤드](#)가 있을 수 있다
- 가상함수를 사용하지 않고 이벤트를 처리할 수는 없을까?

```

1 template<typename T>
2 class Window{
3   public:
4     void event_loop() {
5       static_cast<T*>(this)->Click(); // CRTP 필요함
6     }
7     void Click() { }
8   };
9
10 class MainWindow : public Window<MainWindow>
11 {
12   public:
13     void Click() { }
14   };
15
16 int main() {
17   MainWindow w;
18   w.event_loop();
19 }
```

- 가상함수를 없애고 `event_loop`에서 파생 클래스의 함수를 호출하기 위해서는 `static_cast`를 해야한다
- 이 때, CRTP를 사용해 `static_cast`에 파생 클래스의 이름 `T`를 넣어줘야 한다
- CRTP를 사용하면 가상함수가 아닌 함수를 가상함수처럼 동작하게 할 수 있다

CRTP를 사용할 때 주의 사항이 있다. 기반 클래스 템플릿이므로 파생 클래스의 갯수가 많아지만 [코드 폭발 \(code bloat\)](#) 현상이 발생할 수 있다.

```

1 template<typename T>
2 class Window{
3   public:
4     void event_loop() {
5       static_cast<T*>(this)->Click(); // CRTP
6     }
7     void Click() { }
8   };
9
10 class MainWindow : public Window<MainWindow>
11 {
12   public:
13     void Click() { }
14   };
15
16 // Window 기반 클래스의 여러 함수들이 MainWindow2 타입의 새로운 템플릿 인스턴스로 생성된다
17 class MainWindow2 : public Window<MainWindow2>
18 {
19   public:
20     void Click() { }
```

```

21     };
22
23     ...
24 // 파생 클래스가 많아질수록 새로운 템플릿 인스턴스가 계속 증가한다

```

따라서 T를 사용하지 않는 Window의 멤버 함수는 별도로 빼놓는 방법도 존재한다.

```

1 class BaseWindow { // CRTP와 무관한 함수를 별도 기반 클래스로 분리
2     public:
3         void Click() {}
4         void MouseMove() {}
5     }
6
7     template<typename T>
8     class Window : public BaseWindow
9     {
10         public:
11             void event_loop() {
12                 static_cast<T*>(this)->Click(); // CRTP
13             }
14     };
15
16     class MainWindow : public Window<MainWindow>
17     {
18         public:
19             void Click() {}
20     };

```

- T의 영향을 받지 않는 BaseWindow라는 기반 클래스를 따로 설정하여 코드 폭발을 방지할 수 있다
- 이런 기술의 이름을 **thin template**이라고 한다

CRTP 기술을 사용하는 c++20에 추가된 `std::view_interface`를 살펴보자

```

1 template<typename T> class view_interface {
2     // 자기 자신을 파생 클래스로 캐스팅하는 함수
3     const T& Cast() const { return static_cast<const T&>(*this); }
4     public:
5         bool empty() const {
6             auto& Derived = Cast();
7             return Derived.begin() == Derived.end(); // 파생 클래스의 begin, end가 반드시 구현되어 있어야 함
8         }
9     };
10
11     template<typename T>
12     class transparent_view : public view_interface<transparent_view<T>>
13     {
14         T& range;
15     public:
16         transparent_view(T& r) : range(r) {}
17         auto begin() const { return range.begin(); }
18         auto end() const { return range.end(); }
19     };
20
21     int main() {
22         std::vector<int> v;
23         transparent_view tv(v);
24         std::cout << tv.empty() << std::endl;
25     }

```

- `view_interface` 클래스의 파생 클래스를 만들 때 `empty()` 함수로 인해 `begin`, `end`가 반드시 구현되어야 한다

CRTP를 사용하는 또 다른 예제를 살펴보자

```

1 class too_many_object : std::exception {};
2

```

```

3   class LimitObjectCount {
4     inline static int maxcnt = 0;
5   public:
6     LimitObjectCount() {
7       if(++maxcnt > 5)
8         throw too_many_object();
9     }
10    ~LimitObjectCount() { --maxcnt; }
11  };
12
13  class Player : public LimitObjectCount {
14  };
15
16  class Judge : public LimitObjectCount {
17  };
18
19  int main() {
20   Player p[3];
21   Judge j[3]; // error! 기반 클래스의 static int maxcnt가 공유되기 때문에 총합이 5를 넘을 수 없다
22 }
```

- 서로 다른 maxcnt를 사용하려면 서로 다른 static 변수를 사용해야 한다
- CRTP를 사용하면 서로 다른 템플릿 인스턴스가 만들어져 각각 5개 이상씩 할당받을 수 있다

```

1   class too_many_object : std::exception {};
2
3   template<typename T, int MAXCOUNT = 5>
4   class LimitObjectCount {
5     inline static int maxcnt = 0;
6   public:
7     LimitObjectCount() {
8       if(++maxcnt > MAXCOUNT)
9         throw too_many_object();
10    }
11    ~LimitObjectCount() { --maxcnt; }
12  };
13
14  class Player : public LimitObjectCount<Player, 5>
15  {};
16
17  class Judge : public LimitObjectCount<Judge, 3>
18  {};
19
20  int main() {
21   Player p[3];
22   Judge j[3]; // ok
23 }
```

다음으로 CRTP 기반의 싱글톤 패턴 코드에 대해 살펴보자

```

1   template<typename T>
2   class Singleton {
3     public:
4       static T& getInstance() {
5         static T instance;
6         return instance;
7       }
8   };
9
10  class Cursor : public Singleton<Cursor>
11  {
12   private:
13     Cursor() {}
14     Cursor(const Cursor&) = delete;
15     Cursor& operator==(const Cursor&) = delete;
```

```
16
17     friend class Singleton<Cursor>
18 }
19
20 int main() {
21     Cursor& c = Cursor::getInstance();
22 }
```

- CRTP를 사용해 싱글톤 패턴을 구현할 수 있다. 예전에 많이 사용했던 방법이다

7 References

- [1] (lecture) CODENURI - C++ Master
- [2] (blog) [모던C++] 정리 - tango1202

8 Revision log

- 1st: 2024-07-16
- 2nd: 2024-07-23
- 3rd: 2024-07-30
- 4th: 2024-08-01
- 5th: 2024-08-11
- 6th: 2024-12-13
- 7th: 2025-01-07
- 8th: 2025-01-23
- 9th: 2025-01-25
- 10th: 2025-01-30
- 11th: 2025-01-31
- 12th: 2025-02-02
- 13th: 2025-02-21
- 14th: 2025-02-22